TEMPORAL KNOWLEDGE BASE MANAGEMENT: MODEL, QUERY LANGUAGE, ALGEBRA AND IMPLEMENTATION TECHNIQUES

By

HSIN-HSING M. CHEN

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1993

Dedicated to my wife and our parents.

ACKNOWLEDGEMENTS

I would like to express my sincerest appreciation to my advisor, Dr. Stanley Su, for giving me the opportunity to work on this interesting and challenging research topic and especially for his guidance, encouragement and continuous support during the period of this research. I am also grateful to all the members of my supervisory committee, Dr. Herman Lam, Dr. Jose Principe, Dr. Li-Min Fu, and Dr. Sharma Chakravarthy for their thought-provoking suggestions on this work and their time in serving on my supervisory committee. Last but not the least, I would like to thank our secretary Mrs. Sharon Grant for all the help that she provided during my research.

This research is supported by the National Science Foundation grant# CCR-9200756.

TABLE OF CONTENTS

ACKNOWLEDGEMENT iii

ABSTRACT vi

CHAPTERS

1 INTRODUCTION 1

2 A SURVEY OF RELATED RESEARCH 6

2.1 Temporal Data Models 6
2.2 Temporal Query Languages 7
2.3 Storage Structures and Access Methods 8

3 THE TEMPORAL KNOWLEDGE REPRESENTATION MODEL OSAM*/T 10

3.1 Overview of OO Databases 10
3.2 The Concepts of Time Sequence, Time Granularity and Event 11
3.3 Interval Time-Stamping Strategy 13
3.4 Object Instance Time-Stamping 14
3.5 Using Knowledge Rules to Capture Special Notions of Time 14
3.6 Object Instance History 17
3.7 Object History 17
3.8 Association Histories 18
3.9 Operations for a DBA to Correct Errors 19

4 TEMPORAL KNOWLEDGE RULE SPECIFICATION LANGUAGE AND MODELING TECHNIQUE 23

4.1 Temporal Rules as Part of Knowledge Base Specification 24
4.2 Temporal Rule Specification Language 29
4.3 Managing Temporal Rules 36

5 THE TEMPORAL QUERY LANGUAGE: OQL/T 46

5.1 Temporal Association Patterns 47
5.2 The WHEN Clause 48
5.3 Interval Comparison Operators and Functions 50
5.4 Temporal Functions 53
5.5 Temporal Ordering Functions 56

5.6 Moving Window Functions 57
5.7 Set Operators 60

6 TEMPORAL ASSOCIATION ALGEBRA: A MATHEMATICAL FOUNDATION FOR OBJECT-ORIENTED TEMPORAL KNOWLEDGE BASES 64

6.1 Intensional vs Extensional Temporal Database 66
6.2 Temporal Association Algebra 67
6.3 Temporal Association Algebraic Operators 75
6.4 Specification of Time Domain 89
6.5 Query Example 91
6.6 Comparison with Related Work 94

7 A DELTA-INSTANCE MULTI-SNAPSHOT STORAGE MODEL 111

7.1 Delta-Instance Multi-Snapshot Storage Model 112
7.2 Existing Access Method for Temporal Data 122
7.3 Benchmark Queries 127
7.4 Performance Analysis and Comparison 129

8 SUMMARY 160

REFERENCE LIST 162

APPENDICES

A BNF OF THE OQL/T 169

B BNF OF THE KNOWLEDGE DEFINITION LANGUAGE FOR OSAM*/T 173

C TDB INDEX 180

D ANALYTICAL FORMULAS FOR STORAGE CONSUMPTION 182

E PARAMETERS VALUES USED IN THE ANALYSIS 184

F ANALYTICAL FORMULAS FOR MATERIALIZING TEMPORAL DATA 185

G ANALYTICAL FORMULAS FOR TEMPORAL QUERY PROCESSING 193

H PERFORMANCE EVALUATION AND COMPARISON OF INDEXING TECHNIQUES 205

BIOGRAPHICAL SKETCH 226

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

TEMPORAL KNOWLEDGE BASE MANAGEMENT: MODEL, QUERY LANGUAGE, ALGEBRA AND IMPLEMENTATION TECHNIQUES

By

Hsin-Hsing M. Chen

May 1993

Chairman: Dr. Stanley Y.W. Su
Major Department: Electrical Engineering

There has been a considerable amount of work on object-oriented databases, active databases, and deductive databases. The common objective of these efforts is to produce highly intelligent and active systems for supporting the next generation of database applications. These future systems must be capable of capturing the concepts of time and managing not just temporal data but temporal knowledge expressed by knowledge rules. In this dissertation, we describe our efforts on a temporal object-oriented knowledge model, OSAM*/T, its associated temporal query language, OQL/T, an underlying temporal algebra, TA-algebra, and some implementation techniques. In addition to the features of the traditional object-oriented paradigm, the model is characterized by its strong support of association types and its incorporation of temporal knowledge rules for specifying temporal and other types of semantic constraints associated with object classes and their temporal object instances. The query language is distinguished by its pattern-based specification of temporal object associations, which

allows complex queries with various time constraints to be formulated in a relatively simple way. The temporal algebra provides a set of primitive operators for manipulating homogeneous and/or heterogeneous patterns of temporal object associations, thus providing the needed mathematical foundation for processing and optimizing temporal queries. The implementation techniques include a Delta-Instance and Multi-Snapshot Storage Model, as well as data partitioning and clustering schemes for storage management of temporal knowledge bases. Also, in order to understand the relative merits of the techniques proposed in our approach and those of the existing proposals, we evaluate and compare their performances in terms of storage consumption, time for materializing temporal data and times for processing temporal queries.

CHAPTER 1
INTRODUCTION

Conventional database management systems store and manage only the snapshot of the most recently updated data of an application domain, which is often called the current database. Whenever an update occurs to the current database, old data (or temporal data) are replaced by the new data and are lost forever. Conventional DBMSs do not provide the facilities for recording, storing and processing historical data. Although data with time information (e.g., birthdate of an employee) can be defined as attributes, they are processed and interpreted by users' queries or application programs instead of being maintained and processed by DBMSs.

The importance of managing and using temporal data has been well recognized in the database and AI communities for the past two decades [BOL82, McK86, SNO86, SNO90, MAI91, SOO91, JEN92a]. Temporal data, which are the recordings of past activities, are often useful for present and future decision making in an information system. A database that contains temporal information of entities or objects is called a temporal database. As noted by Bjornerstedt and Hutten [BJO89], temporal data can be used not only to model the history of real-world entities for a wide range of applications, such as VLSI/CAD design, office information systems, scientific databases, multimedia databases, etc., but also to handle the notion of time in several system-level applications, such as concurrency control, crash recovery, etc. For example, Sari et al. [SAR86] used historical information to process delayed database updates; Stonebraker [STO87] used historical information instead of a conventional write ahead log (WAL) for transaction

management in the design of the Postgres storage system. Moreover, Allen [ALL84] and others [DEA86, DE87] used temporal data as the basic assertions for a knowledge base management system to reason about past, current or future activities and to derive facts that are not explicitly stored [ALL84, DEA86, DE87]. Recently, in the Lagunita workshop on future research of database systems, Silberschatz et al. [SIL90] further identified the need for database management systems to support time in meeting the increased complexity of next-generation database applications.

Past research on temporal DBMSs has been limited by the capacity and the speed of secondary storage media. However, the technology in storage media has advanced to a stage that large quantities of temporal data can be stored and processed efficiently [HOA85, SNO86]. This has motivated a considerable amount of research on temporal databases. They include the design of data models for modeling temporal data [CLI83, CLI85, SNO85, GAD86, TAN86, SEG87, LOR88, NAV89, ROS91, SU91, JEN92a, WUU92] and of high-level query languages and their associated algebra for accessing temporal data and optimizing query processing [CLI85, GAD86, TAN86, SNO87, LOR88, NAV89, SAR90, TUZ90, GUO91, McK91, ROS91, SU91], as well as the management of secondary storage to achieve efficient indexing techniques for data access [LUM84, AHN86, AHN88, ELM90a, ELM90b, ELM91]. Most of these efforts have been based on the relational model, which, however, has its limitations in modeling complex objects and application semantics. In spite of all these recent efforts, temporal DBMSs still have not been put into practice because of the concerns of excessive storage consumption and processing inefficiency resulting from data redundancy, excessive requirements of storage space for extra time notions, and computational overhead.

Data redundancy. Data redundancy exists in the tuple time-stamping technique [LUM84, SNO85, LOR88]. In tuple time-stamping, time tags are attached to each tuple

of a relation. When any attribute of a tuple is modified, new time tags will be created for the new version of the tuple, which contains both the modified attribute(s) and the remaining unchanged attributes; the original tuple becomes historical data. Data redundancy thus exists between the new and the old versions of tuples for those unchanged attributes.

Excessive storage requirements. In a temporal DBMS, extra time notions are often introduced to capture unusual application semantics such as proactivity, retroactivity, etc. Adding extra time notions to a temporal DBMS, however, has three disadvantages. The first disadvantage is the requirement of extra storage space to store extra time tags for each data unit and its history. The second disadvantage is the need to predetermine the number and kinds of time notions used in a system. The third disadvantage is the difficulty of retaining a closure property and the possibility of losing information [McK91].

Computational overhead. Computational overhead exists in the attribute time-stamping technique. In attribute time-stamping, temporal data is represented either as Temporal Normal Form (TNF) or nested relations. Since the attributes of a tuple may not always be modified at the same time, the time tags of each attribute in the attribute time-stamping approach are different. Therefore, when the data of an entire tuple of certain time interval are required, temporal joins between the TNF relations or "unnest" operations for the nested relations are necessary. Joining two TNF relations or unnesting nested relations, however, is very time-consuming and costly because each temporal join between two time-stamped attributes consists of one equijoin and two outerjoins [SEG89] and unnesting a nested tuple would require a computation cost with the complexity of the order of $O(M^N)$, where M is the average number of evolutions of each attribute and N is the number of attributes of a tuple.

Motivated by the disadvantages of the existing techniques and the modeling limitations of the relational model, we propose an object-oriented (OO) knowledge base management approach to model and manage temporal data. This approach is taken for the following reasons. First, the use of the OO paradigm for information processing is a current trend. The paradigm offers a number of features, such as property inheritance, object identity, encapsulation, information hiding, and type/class mechanism [BAT84, COP84, BOO86, KHO86, MAI86, ZDO86, KIM89, NIE89, KHO90, ZDO90, KIM90a], that allow real-world entities and their histories to be modeled and processed in a more natural way. Second, using a KBMS to model and manage historical information offers the reasoning capability. And third, the model in our approach is a semantic model, which provides a set of semantic-rich constructs for modeling the relationships among the real-world entities [CHE76, SMI77, HAM78, SU89, ELM90b]. In this dissertation, we investigate the following tasks altogether rather than as separate issues: the design of an OO temporal knowledge representation model for modeling temporal data and temporal knowledge rules, the design of an OO temporal query language for processing temporal data, the development of a temporal association algebra, and the design of a storage model for managing temporal data. By doing so, we believe that we are able to propose better and integrated solutions to the problems of modeling limitation, excessive storage consumption, and processing inefficiency found in a temporal database.

This dissertation is organized as follows. A survey of the existing efforts on modeling, querying and implementing temporal databases is presented in Chapter 2. Chapter 3 presents the conceptual temporal OO knowledge representation model OSAM*/T. Modeling of objects as well as time semantics associated with real-world applications are addressed. In Chapter 4, we present the specification language and modeling technique for temporal knowledge rules. Temporal knowledge rules are also

modeled as objects and treated as part of the class definition. They can be processed just like other data objects and can also be updated as their applicabilities change, thus having histories. In conjunction with the temporal knowledge model, we design a temporal query language called OQL/T for processing temporal data. The concept of association-pattern based query formulation and the temporal functions introduced in OQL/T are presented in Chapter 5. Chapter 6 describes a temporal association algebra (TA-algebra), which provides the mathematical foundation for processing temporal OO knowledge bases. The concepts of intensional and extensional temporal OO knowledge bases from the algebra point of view are presented. In Chapter 7, we present a Delta-Instance and Multi-Snapshot storage model, which is designed to support a large and long-lived temporal knowledge base with a low storage requirement and high processing efficiency. The evaluation and comparison of performance between our proposed storage model and the existing techniques in terms of storage consumption and query processing time are also presented. Lastly, we summarize our research in Chapter 8.

CHAPTER 2
A SURVEY OF RELATED RESEARCH

Existing research on temporal DBMSs focus on three areas: the design of a temporal model, the design of a temporal query language, and the management of secondary storage. A survey of the related work in these three areas is presented in this chapter.

2.1 Temporal Data Models

Most of the existing research in temporal DBMSs has been based on the relational data model. Techniques used to model temporal information using the relational model are attribute time-stamping [COP84, CLI85, GAD86, TAN86, ELM90a] and tuple time-stamping [LUM84, SNO85, LOR88]. The expansion and contraction of data between the normalized and the nested relations required in the attribute time-stamping technique produce a serious time overhead in processing temporal data; whereas, in tuple time-stamping, data-redundancy is the major shortcoming. In order to take advantages of these two techniques and to avoid their shortcomings, Navathe and Ahmed [NAV89] used a combined approach that is based on the concept of "temporal normalization." However, their approach only works under a weak assumption called "synchronous attributes," which is a rare case in the real-world application. If "synchronous attributes" are not available in an application, the approach will become tuple time-stamping over relations that contain only two attributes (i.e., the primary key and a variant attribute). This logical data representation will be worse than using the

tuple time-stamping technique over nonbinary relations because it generates more redundant data (i.e., replication of keys) and needs more join operations during data processing.

In addition to the problems associated with the existing techniques, there are three common problems found in a relational temporal database: (1) the need for more than two time notions, which causes the difficulty of retaining the closure property in query processing, (2) the mixing of historical data with current data, which degrades the performance of processing the current data, and (3) the limited capabilities in modeling complex objects, constraints, and behavioral properties.

2.2 Temporal Query Languages

Most of the existing research in temporal query languages involves the extension of relational calculus-based query languages such as Quel or SQL [CLI85, GAD86, TAN86, SNO87, LOR88, NAV89, ELM90a, KIM90] and the relational algebra [CLI85, GAD86, TAN86, LOR88] using some temporal operators and constructs for the specification of temporal requirements. The temporal extension of a relational query language is often influenced by the time-stamping technique used. For example, with attribute time-stamping, the temporal extension of a query language will require operators such as "pack," "unpack," etc. [CLI85, GAD86, TAN86], to transform temporal data between normalized and nonnormalized forms.

Different approaches have been taken to achieve the temporal extension of a relational query language. For example, Snodgrass [SNO87] proposed TQuel, which aims to be a minimal extension of Quel for meeting simple temporal requirements [NAV89]; Elmasri [ELM90a] focused on the primitive temporal operations such as select and project for an ER model; Kim et al. [KIM90] designed ETQL to support

applications of abstract time. These works are important and interesting. However, the constructs and operators introduced in these extensions are limited and they do not include constructs for a user to express temporal requirements over several snapshots.

2.3 Storage Structures and Access Methods

Existing research on storage structures and access methods for the storage and processing of temporal data has aimed to retain the efficiency of processing current data and, at the same time, to increase the efficiency of processing historical data. The research in this area can be divided into two categories: partitioning of temporal data and indexing of temporal data.

For temporal data partitioning, Lum et al. [LUM84] proposed to divide temporal data into "current store" and "history store" and Ahn and Snodgrass [AHN86, AHN88] proposed five storage structures for storing and partitioning temporal data (i.e, reverse chaining, accession list, clustering, stacking, and cellular chaining). Both proposed partitioning strategies and storage structures allow current data to be processed efficiently. However, the processing of temporal data in Lum's and Ahn's approaches, which are based on the concept of reversed history chain, requires lengthy traversals of history chains, especially when older temporal data are to be processed. Additionally, the cost of traversing a history chain increases as the history chain grows.

For improving the efficiency of processing temporal data, several indexing techniques have been proposed. For example, Gunadhi and Segev [GUN90] proposed AP-tree, nested ST-tree, and nested AT-tree based on the concept of start-time of the valid time notion for processing efficiently those queries involving various searching criteria such as time, attribute values, object identifiers and their combinations; Elmasri et al. [ELM90b, ELM91] proposed a Time-Index and several of its variations based on

the concept of both start-time and end-time of the valid time notion for efficient processing of those queries that use time point as the searching criterion. The proposed indexing techniques, however, have the disadvantage of increasing the cost of searching historical information as a database evolves.

CHAPTER 3
THE TEMPORAL KNOWLEDGE REPRESENTATION MODEL OSAM*/T

This chapter presents the conceptual temporal object-oriented (OO) knowledge representation model OSAM*/T. We first give an overview of the relevant modeling concepts of OO databases. The presentation of the proposed time-related concepts and constructs of OSAM*/T ensues.

3.1 Overview of OO Databases

A temporal OO data model provides the conceptual basis for defining temporal OO databases. OSAM*/T is a temporal OO semantic association model that provides a conceptual basis for uniformly capturing the interrelationships among temporal objects in an application world. An entity, which can be a physical object, an abstract thing, an event, a process, a function or anything that an application cares to define, is modeled as an object and will have a system-assigned object identity (i.e., OID) for its unique identification when it is created initially. Objects having the same structural and behavioral properties are grouped together to form an object class. Each object class has a system-assigned class identifier (CID). An object in a database can participate in more than one class, and the data representation of the object in a class is called an object instance. An object instance in a class is uniquely identified by an instance identifier (IID), which is the concatenation of CID and OID (that is, IID = CID || OID). Object classes can be categorized into two general categories: (1) the entity-class, which represents a set of objects of interest in an application world, each of which is

assigned a system-wide unique object identifier (OID) and its data are explicitly entered in a database by a user, and (2) the domain-class, which represents a class of self-naming objects whose values identify the objects and serve as a domain for defining other object classes (e.g., a class of symbols or numerical values). The behavioral properties of an object class are defined in terms of system-defined or user-defined operations (e.g., retrieve, display, delete, insert, rotate a design object, hire an employee, etc.), which can meaningfully operate on its object instances using their corresponding programs (or methods). The structural properties of an object class consist of two types of data that define the states of its object instances: (1) descriptive data (or instance variables) and (2) association data, which specify the relationships between its object instances and the object instances of some related classes. Object classes and their instances are then interrelated through various association types, each of which can be defined by a set of operational rules governing the manipulation of object instances of the associated classes. In addition to the two most commonly recognized associations, Aggregation and Generalization, OSAM*/T supports three other association types (i.e., Interaction, Composition, and Crossproduct). Additional user-defined association types or subtypes of these existing types are possible since association types are modeled as classes in the implementation model [YAS91].

3.2 The Concepts of Time Sequence, Time Granularity, and Event

OSAM*/T uses a discrete time model for time representation in which time is viewed as being isomorphic to natural numbers and is represented as a time sequence. A time sequence is a series of discrete time points t_1, t_2, ... t_{n-1}, and t_n in which the relationship of a total ordering between two time points is obeyed: $t_1 < t_2 < ... < t_n$.

The distance between any two adjacent time points is identical and is called a time unit. That is,

$$\text{time unit} = t_2 - t_1 = t_3 - t_2 = \ldots = t_n - t_{n-1}.$$

Time unit may be different from application to application depending on the granularity needed. It can be microsecond, minisecond, second, minute, hour, day, month, or ..., etc. It is also possible that a system may use more than one time granularity for its applications' needs. Under this situation, functions for the conversion between different time granularities must be provided. In the rest of this dissertation, we shall use just numerical values to capture the notion of time granularity as is intended in each particular application for reasons of simplicity without losing the significance of different time granularities. We shall also assume that the history of object instance is recorded discretely; however, the interpretation of an object history is continuous.

An event is an action that will cause a change in the contents of a database. An update, a delete, an insert, or a disassociate operation is an event. For example, increasing John's salary to $30,000, deleting Mary's record, inserting Brown's record, and disassociating Tom's relationship with a particular project are events. A data retrieval operation is not considered an event because it does not alter the state of a database and, thus, does not affect the time tags associated with the data. The relationship between time sequence and event is illustrated in Figure 3-1.

In a temporal database, the only time an object instance has a new time tag and a new instance value is the time when an event occurs (i.e., any of its attributes or association is modified, deleted, etc.) The old instance then becomes a historical version and a part of the history of this object instance. A historical version of an object instance in a temporal database is stored in a historical area and can be uniquely identified by TIID (temporal IID), which consists of the valid time interval and the IID

associated with this historical object instance. For example, to adjust John's salary to $30,000 beginning at time 8 will introduce a new instance to replace the old one in the current database and the old instance will become history and be shifted into the historical area (the instance of employee John in this example is modeled by the instance identifier 012):

Start-time	End-time	IID	Title	Dept	Salary	
< 8,	- ,	012,	Engr	R&D	$30,000> . . .	New instance.
< 1,	7,	012,	Engr	R&D	$20,000> . . .	Old instance.

The concept of evolution in the time dimension for this object instance can be illustrated by Figure 3-2: during T[1,7], object instance 012 is associated with Engr title, R&D dept, and $20K salary; during T[8,NOW], 012 is associated with Engr title, R&D dept, and $30K salary.

3.3 Interval Time-Stamping Strategy

In OSAM*/T, we adopt the interval time-stamping strategy, which uses Start-time and End-time tags of the valid time notion to record and delimit each historical instance. The Start-time is the time when the data about an object become true. Creating an object, inserting an object instance, or updating an object instance will cause a new Start-time to be attached to the object instance having new attribute values. The End-time is the time when the data about an object has been changed in the next time unit. For example, the operation of destroying an object in a database, deleting an object instance from a class, or updating an object instance in a class will cause an End-time to be attached to the object instance which is about to be modified.

3.4 Object Instance Time-Stamping

The object instance time-stamping technique proposed in this research is a compromise between the tuple time-stamping approach and the attribute time-stamping approach used in many existing works. The time stamps in our approach are assigned to a set of semantically related attribute values that represent an instance of an object class. Every object instance is time stamped with a time interval, <Start-time, End-time>. Since in our model, an object can participate in multiple classes and have multiple object instances, not all the data of an object have the same time stamp as in the tuple time-stamping approach. If an object instance is in the current database, we say that the object instance is currently active; otherwise, it is inactive. Active object instances have infinite End-time represented as " - ". When an active object instance is affected by an event, its End-time is set to one time unit before the event time, and the object instance becomes a historical instance and is shifted into the historical area. A new active instance is then created with Start-time set to the event time and End-time set to "-". The example that records the change of John's salary given in section 3.2 is an example of object instance time-stamping.

3.5 Using Knowledge Rules to Capture Special Notions of Time

It has been suggested that extra time notions in addition to the valid time be introduced to capture temporal requirements for some particular applications such as retroactivity [LUM84, SNO85, NAV89]. These time notions include transaction time, effect time, physical time, logical time, or any other user-defined time notions. Adding extra time notions to a temporal DBMS however has three disadvantages. The first disadvantage is the requirement of extra storage space: once a time notion is introduced into a database, extra storage space is needed to store the time notion for each data unit

(i.e., object instance, tuple, or attribute) and its history. The second disadvantage is the need to predetermine the number and types of time notions used and supported in a system. Once the number and types are fixed, any later addition of extra time notion would require a drastic change of database and DBMS. The third disadvantage is the difficulty in retaining the closure property and the possibility of losing information due to operations such as JOIN and PROJECT [McK91]. Therefore, it is important to keep the number of time notions small.

We observe that most of the real world applications only concern the valid time of data and that some time notions are applicable only to some specific data in a specific application. For example, the fact that "the employee Mary's salary, $30K, is retroactively effective (at time 10) from time 7 instead of from time 8" only affects the object instance Mary, and it should not be treated as a general case. Therefore, in OSAM*/T, we use only two general time tags of the valid time notion, Start-time and End-time, for recording the histories of object instances. The semantics of the other time notions are expressed by knowledge rules. Using knowledge rules to capture extra time notions for some particular applications has the advantages of achieving storage-saving, flexibility in introducing extra time notions, and easier retention of the closure property for database operations. We shall use the retroactive update problem as an example to show the advantage of using knowledge rules over an extra time notion.

Approach 1. Introduce a "Record-time" time notion to capture the fact that Mary's salary of $30K has been retroactively effective (at time 10) from time 7 (where the object instance of employee Mary in this example is modeled by the IID 011). The original database and the updated database are given below:

(1) Original database

Record-time	Start-time	End-time	IID	Title	Dept	Salary
< 8,	8,	-,	011,	Sect.	Sales	$30,000>
< 3,	3,	7,	011,	Sect.	Sales	$20,000>

(2) Updated database

Record-time	Start-time	End-time	IID	Title	Dept	Salary
< 8,	8,	-,	011,	Sect.	Sales	$30,000>
< 10,	7,	7,	011,	Sect.	Sales	$30,000>
< 3,	3,	7,	011,	Sect.	Sales	$20,000>

In this approach, we note that the extra time notion "Record-time" is used to solve the problem of recording the fact that Mary's salary is retroactively updated. However, since the Record-time is in the database schema, it also introduces excessive storage requirements because every object instance and its history in the database will need the Record-time tag.

Approach 2: Use a Knowledge rule (instead of introducing a "Record-time") to capture this particular application for object instance Mary.

(1) Original database (Note: No Record-time in this case)

Start-time	End-time	IID	Title	Dept	Salary
< 8,	-,	011,	Sect.	Sales	$30,000>
< 3,	7,	011,	Sect.	Sales	$20,000>

Rule 105
Valid_T[10,-]
Triggered (Before RetrieveSalary())
condition (INTERVAL(this) = T[7,7])
action derived_value (this.Salary, $30K)
End

In this approach, the knowledge rule which is pertinent only to the object instance Mary is defined to capture the fact that Mary's salary of time 7 is retroactively updated at time 10 from $20K to $30K. Whenever Mary's salary is to be retrieved, this rule will be triggered. If it is detected by the system (through checking the condition "INTERVAL(this) = T[7,7]") that Mary's salary of time 7 is being requested, Rule 105 will be fired and the method "derived_value (this.Salary, $30K)" specified in the action-clause of this rule will be executed to reflect Mary's retroactive salary $30K. The computation caused by the method derived_value(...), however, does not change the

content of the database. The cost in this case is the storage and maintenance of Rule 105, which is much cheaper than the enormous amount of space needed for storing many extra time tags for all data units in the first approach. This knowledge rule can also be deactivated by some high-level language constructs for some other applications.

We note that in this knowledge rule, the keyword "this" is bound to the object instance which is being processed and the temporal function INTERVAL() is used to retrieve the interval of a historical object instance. Knowledge rules provided in OSAM*/T can be used to capture not only special time notions, but also constraint and deductive knowledge in a KBMS. Detailed discussion of knowledge rules in OSAM*/T will be given in Chapter 4.

3.6 Object Instance History

In OSAM*/T, an object can participate in many classes. The descriptive data about an object are distributed in these classes. An instance is the representation of an object in a specified class and contains the attribute values that characterize the object. When any of these attribute values is modified, a history of the object instance is created. If an object participates in more than one class, each object instance in each class will have its own history.

3.7 Object History

Object history can be viewed as the collection of the histories of its instances in the classes in which the object participates. Deletion of any of its instances from a class only means the withdrawal of its participation in that class and does not affect other instances. When an object is created, it must participate in at least one class in the schema. This participation starts the object history as well as the object instance history

in that class. If the object participates in only one class, then the object instance history is the same as the object history. When an object is destroyed (or deleted permanently from a database), all the instances of this object will have nulls for their attribute values. But, the identity of that object is still recognized by the system.

3.8 Association Histories

In OSAM*/T, object classes are defined in terms of their associations with other classes. Five system-predefined association types (Aggregation, Generalization, Interaction, Composition, and Crossproduct) are provided for the convenience of database designers and users to specify different semantic relationships or associations among classes and their instances. The semantic distinctions among the association types are captured by knowledge rules that control the manipulations of object instances having the association types. In our KBMS implementation, association types are defined as classes that are subclasses of the Association class. Thus, new association types or subtypes of the existing association types can be introduced by defining subclasses under the Association class or its subclasses. It has also been recognized [ELM90a] that, besides object history and object instance history, the history of object associations needs to be maintained. One of the advantages of object instance time-stamping is that association histories among object instances can be derived from object instance histories. Whenever an event occurs to an object instance, the corresponding association history can be inferred from that instance history. We shall use the Interaction association as an example to illustrate this point.

Interaction is an association type used to model some relationships between object instances in two or more classes; the relationships themselves are treated as instances of an object class. An object instance in a defining class that has an interaction

association with other classes (called constituent classes) represents a fact that relates the object instances of its constituent classes. For example, the fact that an employee works on a specific project will form an object instance of a defining class Work_On as modeled in Figure 3-3(a). Since the fact is modeled as an object in Work_On class, it is assigned with a unique IID. Because associations are themselves objects, the tracking of the history of an interaction association is the same as that of other objects. For example, if Mary works on P1, an interaction object instance of Work_On will be created and be assigned with an IID, W1, to indicate the interaction association between Mary and P1. This object instance will consist of Mary and P1's IIDs in addition to its own properties and IID. Any change on either Mary's attributes or P1's attributes will not affect this interaction association. However, a deletion of either Mary's instance or P1's instance from Employee or Project class or Mary's withdrawal from P1 will cause the interaction instance to be deleted (referential constraint). In either case, nulls will replace Mary and P1's IIDs in W1's new version as shown in Figure 3-3(b). In W1's history, the nulls in the historical record with the time interval between 7 and 10 indicate that the interaction association between Mary and P1 does not exist during that period. From time 11 on, Mary has been working on project P1 again. This is represented by storing Mary and P1's IIDs in the object instance W1. The management of the histories of other association types can be similarly handled.

3.9 Operations for a DBA to Correct Errors

In a temporal database, errors in object history due to careless data entry is unavoidable and should be corrected. Corrections of these errors by updates or deletions should be allowed without treating these operations as temporal events. Examples of these errors are mistakes of employee's salary such as $27K for $37K, or

$10K salary increase for no increase at all. For the first case, the record should be corrected while, for the second case, the historical record should be deleted.

CorObj and DelObj are the two operations provided in OSAM*/T for a DBA to deal with the above cases. CorObj will correct the data of a historical object instance. Only one object instance can be corrected at a time; however, multiple attributes of an object instance can be corrected at the same time. The syntax for the use of the CorObj is CorObj((att-i, ...),(newdata, ...)), where att-i are the attributes to be corrected and the newdata are the correct data. DelObj will delete a historical object instance. When an object instance is deleted, the lifespan(s) of the adjacent object instance(s) in the historical database needs to be adjusted correspondingly. For example, suppose in the original temporal database, employee e1 has salaries $30K, $40K and $42K during the time intervals T[1, 3], T[4, 11], and T[12, 16], respectively. If it is found later that the salary raise for e1 during [4, 11] is not true, then the version of e1 during T[4, 11] needs to be deleted and the time interval of the instance of the previous version (i.e., $30K during T[1, 3]) will be adjusted to T[1, 11].

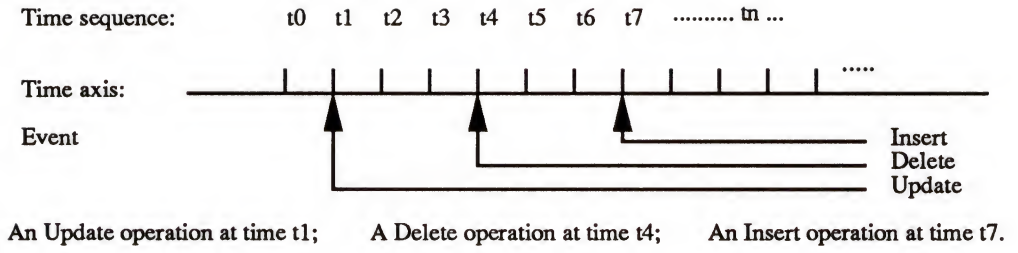


Figure 3-1: Illustration of a time sequence and events

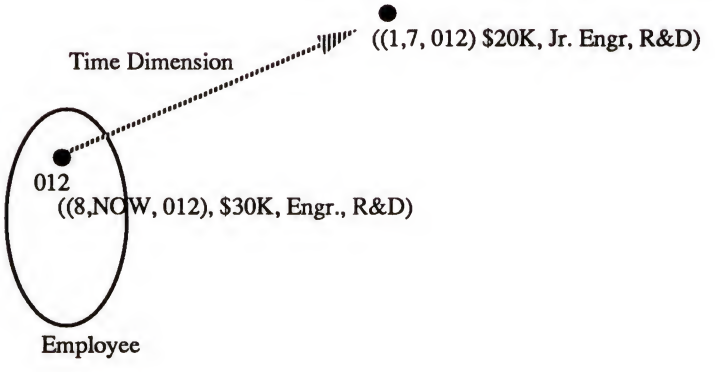


Figure 3-2: The evolution of object instance John in the time dimension.

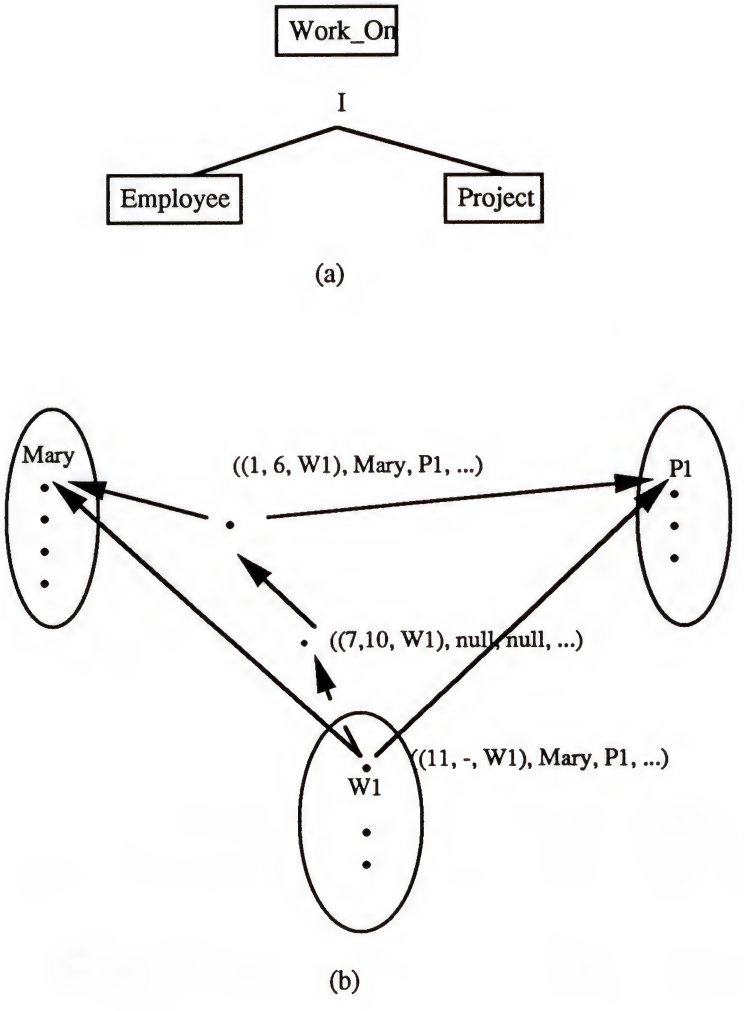


Figure 3-3: Illustration of Interaction history.
a) The Interaction relationship between Employee and Project classes is modeled by Work_On class; b) History of the object instance W1 in the Work_On class.

CHAPTER 4
TEMPORAL KNOWLEDGE RULE SPECIFICATION LANGUAGE AND MODELING TECHNIQUE

As a knowledge representation model, OSAM*/T uses knowledge rules to capture temporal and other semantic constraints of an application domain [SU92]. The knowledge rules of OSAM*/T are called temporal knowledge rules (or temporal rules for short) because they can make reference to temporal data conditions as constraints that are related to past activities. Temporal rules are modeled as conceptual objects and are treated as part of an object class definition. Modeled as objects, temporal rules can be updated as their applicabilities change. As pointed out by Snodgrass [SNO90], it is important for a system to keep histories of temporal rules because they are useful information to applications. In OSAM*/T, the evolution of temporal rules is captured by the object instance time-stamping technique as introduced in Chapter 3. Historical versions of temporal rules can also be triggered when reasoning or making logical deduction on historical data is needed. Triggering of temporal rules and their historical versions depends on their validity period.

This chapter presents the use of temporal rules as part of the object-oriented (OO) knowledge base definition. It also describes a temporal rule specification language with examples to illustrate three general types of temporal rules. Lastly, this chapter discusses the techniques for modeling and processing temporal knowledge rules.

4.1 Temporal Rules as Part of Knowledge Base Specification

Temporal rules in OSAM*/T are treated as part of an object class definition. As introduced in Chapter 3, an object class in OSAM*/T consists of three parts: (1) a specification part, which defines the structural properties of the class (i.e., its descriptive attributes and associations with other classes), the meaningful operations (signatures) that can be performed on its object instances, and the temporal rules that are applicable to its instances, (2) an implementation part, which contains the methods or program code for carrying out the specified operations, and (3) an extension part, which contains the set of object instances belonging to the class. Temporal rules in OSAM*/T define the temporal constraints that objects of a class should satisfy or obey. Temporal rules that are applicable to objects in multiple classes are defined in a superclass having these classes as its subclasses. These rules are their common semantic properties and thus are inheritable by their object instances in a manner similar to the inheritance of common attributes and operations. This category of rules is called class rule. Different temporal rules can also be defined specifically for different instances of an object class. This is achieved by storing these rules as values of a common attribute whose data type is Rule. This category of rules is called instance rule. Thus, a temporal rule can be associated with an individual object instance or with a class (if it is applicable to all the instances of the class). Distinguishing instance rule from class rule has two advantages. First, when some rules are applicable only to some specific object instances, the specification of instance rules is the mechanism for capturing the individual behaviors of the object instances and the processing of other instances will not involve the checking of these rules. Second, the instance rule mechanism allows different sets of rules to be associated with different instances of a class by using several attributes of type Rule. During run-time, a method can explicitly activate or deactivate different sets of rules. For example,

one can search all instances of a class that satisfy some conditions and activate/deactivate a specific set of instance rules. Naturally, the set of class rules applicable to all instances will also be processed.

4.1.1 Object Class Definition

The template of a class definition is shown below:

```
Class_type <class_name>
{ ASSOCIATION_SECTION:
        Association-type = Association_type_1;
                { association_name_1 : domain;
                  association_name_2 : domain;
                ..... }
        Association-type = Association_type_2;
                .....
  OPERATION_SECTION:
        { operation_1();
          operation_2();
          .....}
  TEMPORAL_RULE_SECTION:
        { temporal_rule_1;
          temporal_rule_2;
           ..... }
}/* end of class definition */
```

In OSAM*/T, the structural properties of an object class are defined in terms of its associations with other related classes. As shown in the template, the association section specifies the different system-defined or user-defined association types that an object class has with some other classes. The operation section specifies the operations that are applicable to the instances of the object class. These operations are defined by function and/or procedure declarations (i.e., the signatures) and their methods or program code. The temporal-rule section specifies a set of temporal class rules that are applicable to all of its instances.

In the following, we give an example of a class definition and a class rule. Additionally, we also define an attribute Erule of type Rule (i.e., an aggregation

association between Employee and Rule) in this class definition for a later explanation of instance rule following this example.

Example of an object class definition and a class rule. We define the Employee class based on the semantic diagram (or S-diagram, which is used in OSAM*/T to graphically represent a database schema) shown in Figure 4-1:

```
ENTITY_CLASS Employee
{
ASSOCIATION_SECTION:
        AGGREGATION OF
        { salary: Salary;
          title : Title;
          Erule  : Rule;}
        GENERALIZATION OF
        { Engineer, Manager, Secretary; }
END ASSOCIATION_SECTION

OPERATION_SECTION:
        {TCOUNT(); AVERAGE(); FireEmployee();}
END OPERATION_SECTION

TEMPORAL_RULE_SECTION:
        {
        Rule 00001
        Valid_T [11,-]
        triggered       (before TransferEmployee())
        condition       (exist this in this * Work_On * Project[P# = P2])  ^
                        (exist this in WHEN T[3,5]
                                this * Work_On * Project[P# = P1])
        action          abort
        END /* end of class rule */}
END TEMPORAL_RULE_SECTION
} /* end of class Employee */
```

In this example, a temporal rule is defined in the Employee class with a rule ID 00001. Rule 00001 was defined at time 11 and is still valid (this fact is represented by the valid time interval expression Valid_T [11, -], where Valid_T is a reserved keyword and "-" of the valid time interval stands for an infinite time point). This rule is applicable to all the instances of the Employee class and prevents those employees who are working on project P2 and ever worked on project P1 during the period of T[3,5]

from being transfered. In this rule, the current and temporal conditions which need to be tested for an employee before s/he is transfered are expressed in two quantified expressions which are connected by a logical "and" operator (represented by the caret character " ^ "). Each quantified expression in the condition-clause is delimited by a pair of parenthesis for clarity. In these two quantified expressions, "this" is a keyword which refers to the object (i.e., employee) to be operated on by the operation TransferEmployee(). Each expression checks the existence of this object's association with a Work_On object and a Project object having a specific P#. The time constraint of the first expression is now and the time constraint of the second expression is T[3,5][1]. If the two expressions in the condition-clause are both evaluated to True, the condition-clause is evaluated to True; otherwise, the condition-clause is evaluated to False. When the condition-clause is True, i.e, the employee being transferred is working on P2 and worked on P1 during T[3,5], "this" employee should not be transferred and the "abort" specified in the action-clause will prevent the execution of the operation TransferEmployee(). The expression "this*Work_On*Project[P# = P2]" is an association pattern specification used in our implemented OO query language OQL [ALA89]. It returns all the objects of Work_On class and those Project objects having P# = P2 which are associated with the object identified by "this" as specified by the associate operator. Further explanation of rule format is provided in Section 4.2.

Example of instance rule. Since Employee class has an attribute Erule, every instance of the class can be associated with a different instance rule. In OSAM*/T, the data type Rule is modeled by a class called Rule. An instance rule associated with an

---

[1]In our implementation of the K knowledge base programming language (KBPL) [ARR92, SHY92], the usage of the existing quantifier "exist" follows the syntax: "exist variable(s) *in* context expression [*suchthat* boolean expression], where *in* and *suchthat* are keywords; the keyword (or instance variable) "this" is used to identify the object instance to which an operation (i.e., TransferEmployee() in this example) is being applied.

employee instance is an instance of the Rule class and the value of Erule of the employee instance is the IID of this rule instance. In the following, we give an instance rule for employee John. This rule specifies the formula used to calculate John's actual Salary during the period T[3,6].

Rule 00002
Valid_T [15,-]
triggered (before RetrieveSalary())
condition (INTERVAL(this) = T[3,6])
action derived_value (this.Salary = this.Salary * 1.2)
END /* end of instance rule */

This instance rule identified by 00002 was defined at time 15 for the object instance John of the Employee class and is still valid (represented by the Valid_T[15,-]). It captures the application of retroactive update on John's salary during the period T[3,6] and will be triggered before a retrieval of John's salary during this period. Other operations will not cause it to be verified. Rule 00002 says that John's actual salary of the period T[3,6] should be 20% more than the recorded salary and this retroactive update is valid since time 15. In this example, if the valid time interval of John's historical instance is T[3,6] (specified by the expression "INTERVAL(this)=T[3,6]", where INTERVAL() is a function used to retrieve the valid time interval of the historical object instance), the action "derived_value (this.Salary, this.Salary * 1.2)" specified in the action-clause of this rule will be executed to reflect John's retroactive salary. However, the computation "this.Salary * 1.2" caused by the method derived_value(...) does not change the content of the database.

From the above two examples, it is obvious that both instance rules and class rules have the same format. They can be handled uniformly by the same rule processing mechanism of a KBMS. Definition of temporal rules will be formally introduced in Section 4.2.

4.1.2 Inheritance of Structures, Operations and Rules

The structural and behavioral properties of a superclass can be inherited by its subclasses so that these properties do not have to be defined in the subclasses repeatedly. Inheritance is implied by the "generalization" association in OSAM*/T. For example, the attributes Salary and Title, the operations, and the temporal rules of the Employee class defined in the previous section can be inherited by all its subclasses Engineer, Manager, and Secretary.

4.2 Temporal Rule Specification Language

The general format of a temporal rule in OSAM*/T is given below:

```
Rule rule-id
Valid_T [A, B] /* valid time interval */
triggered (Trigger-time, Trigger-operation)
Rule body
        condition condition-clause
        [action action-clause]
        [otherwise otherwise-clause]
End
```

In this format, "rule-id" is the rule identification (RID). A rule is valid during "Valid_T[A,B]," which is a reserved keyword representing a valid time interval between Start-time A and End-time B; the Start-time is the time when a rule is valid to the temporal KBMS, while the End-time is one time unit before a rule terminates its validity. "triggered (Trigger-time, Trigger-operation)" specifies the situation (or events) in which the temporal rule will be triggered. The "Trigger-time" is drawn from four possible situations: before, after, delayed-after, parallel; the "Trigger-operation" specifies either system-defined or user-defined operations such as InsertInstance(), InsertObject(), DeleteInstance(), TransferEmployee(), etc. The trigger time specifies the time for evaluating the rule relative to the time for carrying out the trigger operation. For

example, the delayed-after delays the processing of the rule body until the end of the transaction of the specified operation. The trigger time of a rule is also the mechanism used in our implementation model for the coupling/decoupling between the original transaction and the spawned transactions from fired rules to achieve modularity as suggested in Dayal et al. [DAY88b]. "Rule body" is expressed by a "condition-clause," an optional "action-clause" and an optional "otherwise-clause". The condition-clause specifies the conditions in terms of boolean expressions, which are to be evaluated when a rule is triggered; the action-clause and the otherwise-clause (one of them can be optional) specify the alternative knowledge base states to be maintained or operations to be performed by the system if the condition-clause is evaluated to True or False, respectively.

In addition to logical expressions that contain logical "and" and/or "or" operators, the condition-clause of a temporal rule can be specified in a guarded expression consisting of a string of expressions of the format "exp1, exp2, ..., expN-1 | expN," in which exp1 to expN-1 serve as the guards for expN. The result of evaluating a guarded expression will be one of the following values: True, False, or Skip. In its evaluation, the first N-1 expressions are evaluated sequentially. If they are all true and the Nth expression is also true, the guarded expression (and thus the condition-clause) is True and the action-clause of the rule will be taken. If they are all true except the Nth expression, the guarded expression (and thus the condition-clause) is False and the otherwise-clause of the rule will be taken. If any of the first N-1 expressions is false, the condition-clause is evaluated to Skip and the action-clause and the otherwise-clause will be skipped (i.e., the rule will not be fired). The mechanism of a guarded expression allows the interdependency relationship among the guards to be explicitly specified. Also, it allows the skipping of the entire rule if any of the guards is false. We note that

the semantics of the sequential evaluation of the guards and the skipping of the entire rule if any one of them is evaluated to false are quite different from logically ANDing the expressions. This is because, due to the optimization of evaluating expressions, the conjunction of these guards do not guarantee their sequential evaluation in a proper order. We also note that, although the semantics of a guarded expression can be specified by a nested if-then-else expression, the former is a simpler and more declarative way of specifying a string of ordered conditions. As noted in [DAY88a, ULL91], the declarative property is an important feature of a high-level language.

4.2.1 Temporal Rules

For the convenience of explanation and illustration, temporal rules in OSAM*/T are classified into three types: temporal state rules, temporal operational rules, and temporal deductive rules [ALA89, CHU90, SIN90, SU91]. Temporal state rules are used to specify legitimate or illegitimate states of a temporal knowledge base. A state rule verifies the state of a knowledge base but does not alter or amend the state of the knowledge base or cause any external operation to take place. Temporal operational rules are used to perform an operation under various temporal conditions (or states). The operation specified in an operational rule will either alter the state of a knowledge base by a system-defined or user-defined operation or cause an external event to occur such as triggering alarm, outputing message to a monitor, etc. by a user-defined operation. Both state and operational rules are used to verify and maintain the correct state of a knowledge base according to some application constraints. Temporal deductive rules, on the other hand, are used to deduce objects' data values and object associations which are not explicitly stored in the knowledge base.

We adopt from our own implementation of K the concept and technique of "object instance binding" in the specifications of temporal rules. If multiple occurrences of one class name in a temporal rule are intended to be bound to the same object instance(s), an instance variable will be defined for the first occurrence of this class name. For example, the expression "e:Employee" will assign the set of object instances of the Employee class to the instance variable e. Each of the succeeding occurrence of the bound class Employee can then be represented simply by the instance variable e. However, if a binding restriction is not intended among the multiple occurrences of a class, each occurrence represents a different scan of the set of instances of the class.

4.2.2 Temporal State Rules

A temporal state rule in OSAM*/T represents a user-defined semantic constraint. It states how the knowledge base activities should be constrained by the past and/or the current activities to ensure semantic consistency and correctness of a knowledge base. A temporal state rule usually involves the verification of more than one knowledge base state which are associated with one or more than one knowledge base activity. The knowledge base states to be verified can be expressed in either guarded or simple logical expressions.

When a knowledge base evolves due to an update, insert, or delete operation, the state of the knowledge base changes. The change will trigger relevant temporal state rules to verify the consistency and correctness of the knowledge base. If any violation against the triggered temporal state rules is detected, the system will abort the operation to avoid the inconsistency. Temporal state rules are useful for a KBMS to verify and maintain legal knowledge base states and to automatically enforce application constraints. The users can therefore be relieved from writing tedious application

programs to enforce the constraints. A temporal state rule defined in the Employee class is given in Example 4- 1.

Example 4-1: Those employees whose salaries are greater than $60K, who are working on project P7, and who worked for the Toy Department during the period T[2,4] should also work for Sales Department.

```
Rule 00004
Valid_T [7,-]
triggered       (After UpdateEmployee())
condition       (this.salary > $60K),
                (exist this in this * Work_On * Project[P# = P7]),
                (exist this in     WHEN T[2, 4]
                                   this * Department[Name = "Toy"])    |
                (exist this in this * Department[Name = "Sales"])
otherwise       abort
End
```

This rule specifies a knowledge base constraint on current employees based on the conditions of their current salaries, current activities, and past activities during the time period T[2,4]. These conditions are specified by a logical expression consisting of three guards and one guarded expression. This rule was defined at time 7 with a user-defined RID "00004" and has been valid ever since; the validity of the rule is represented by the valid time interval Valid_T [7,-]. It will be triggered after an employee is updated.

The triggering condition in triggered-clause specifies that after an employee's record is updated (e.g., updating the salary with 10% increase), the rule should be triggered to check whether the updated record of the employee instance satisfies the conditions stated in the guards of the condition-clause (i.e., whether the employee's salary is greater than $60K, the employee is working on P7 and the employee ever worked for the Toy department during T[2,4]). If so, we want to make sure that this employee also works for the Sales Department (which is expressed by the last expression) before the update transaction can be committed. If it is not true that this employee is also working for the Sales department, the transaction will be aborted because this update will result in a temporal state which violates the application

constraint. Since we only need to check for the employee instance which is being updated, we use the existing quantifier "exist" and the keyword "this" to indicate that the evaluations of the expressions will be limited to the employee instance being updated.

4.2.3 Temporal Operational Rules

A temporal operational rule performs an operation under some data conditions. When a temporal operational rule is triggered, the data condition specified in the condition-clause will be verified to determine whether the action specified in the action-clause or the one in the otherwise-clause should be performed. The operation can be either a system-defined or a user-defined operation. A temporal operational rule defined in the Employee class is given in Example 4-2.

Example 4-2: When an employee is assigned to project P5, (s)he should also be assigned to P6 if (s)he is currently not working for P6 and ever worked for P1 during T[4,12].

```
Rule 00005
Valid_T [17,-]
triggered     (After InsertObject(Work_On))
condition     (exist this in this AND (*that*Project[P# = P5],
                                        !Work_On*Project[P# = P6])) ^
              (exist this in WHEN T[4, 12]
                          this * Work_On * Project[P# = P1])
action        associate(this * Work_On * Project[P# = P6])
End
```

This rule defined in the Employee class will be triggered after a Work_On instance is inserted. The Work_On instance is linked to an employee instance and a project instance (i.e., each instance of Work_On specifies that an employee works on a particular project.) The rule is used to ensure that the present employee (captured by the keyword "this") who is being assigned to project P5 through the inserted Work_On

instance (identified by the keyword "that[2]"), who are not currently working for project P6, and who ever worked on project P1 during the period T[4,12] should also be assigned to project P6. The AND condition following Employee specifies that the Employee instance must be associated with the project instance P5 (the associate operator *) and must not be associated with the project instance P6 (the non-associate operator !). Since we are concerned only with the employee who is being assigned to P5 through the currently inserted Work_On instance and who is not working on P6 in this example, we bind the employee by the keyword "this". With this binding, the checking of the temporal conditions will be done only for the employee instance which is being associated with the inserted Work_on instance. This rule was defined at time 17 with a user-defined RID "00005" and is still valid. The operation, which associates the bound employee with project P6, specified in the action-clause will be executed if condition-clause is evaluated to True.

4.2.4 Temporal Deductive Rules

A temporal deductive rule can be used by a KBMS to deduce/infer a data value for an object or an object association, which is not explicitly stored, based on some temporal information. In a temporal deductive rule, the temporal information which will be used to deduce new fact is expressed in logical expressions in the condition-clause; the deduced data value or object association is expressed in the action-clause or otherwise-clause. The statements in the condition and action/otherwise clauses of a temporal deductive rule are similar to the statements of the condition and consequence

---

[2]"that" is the keyword used in K to bind a presently processed instance which however is not an instance of the class where the temporal rule is defined. For example, Rule 00005 is defined in Employee class and will be triggered by the insert operation of an instance of the Work_On class. In this case, we need to use the keyword "that" to bind the presently inserted Work_on instance in the rule specification.

of a production rule in a conventional expert system. The effect of a temporal deductive rule can be on the past and/or the current state of a knowledge base. Temporal deductive rules for deducing object's data value and object association are given in Example 4-3 and 4-4, respectively.

Example 4-3: Those employees whose present salaries are greater than $60K and who ever participated in project P1 during T[4,12] must be a Senior Engineer.

```
Rule 00006
Valid_T [18,-]
condition     (exist e in e:Employee [Salary > $60K]) ^
              (exist e in WHEN T[4,12]
                        e * Work_On * Project[P# = P1])
action        derived_value(e.title := "Senior Engineer")
End
```

Example 4-4: Those teachers who taught courses of the Computer Science Department during the period T[3,15] must be affiliated with the Computer Science Department then.

```
Rule 00007
Valid_T [18,-]
condition     exist t in (WHEN T[3, 15]
                        t:Teacher * Teach * Course * Dept[Name = "CS"])
action        derived_association ( WHEN T[3, 15]
                                    t * Dept[Name = "CS"] )
End
```

Example 4-3 concludes that those employees whose present salaries are greater than $60K and who ever participated in project P1 during the period T[4,12] must have a (derived) title Senior Engineer as specified in the derived_value() method. In this example, we assume that Title is a derived attribute which does not have an explicitly stored value; instead, its value can only be derived by triggering a temporal deductive rule. The temporal deductive rule of Example 4-3 will be triggered when a query makes reference to an employee's title. Example 4-4 concludes that a teacher was associated with the CS department during T[3,15] if the teacher taught courses offered by the Computer Sciences Department then. In this example, we assume that there is no direct association between classes Teacher and Department in the schema. Their association

can only be inferred from the associations among the object classes Teacher, Teach, Course, and Department. Rule 00007 will be triggered when a query makes reference to the direct association between Teacher and Department classes which does not exist in a schema. In Rule 00007, the method derived_association() is used to derive an object association as specified in its parameter.

4.3 Managing Temporal Rules

Temporal rules are defined in class definitions in the database schema as explained in Section 4.1. Each rule can be parsed and stored in the form of a tree structure similar to a query tree using the same technique used in our earlier implementation of non-temporal knowledge rules [QIU88, CHU90, SIN90]. The information of the tree is then kept in a Rule Descriptor Table and a Rule Body Table of the data dictionary of the KBMS. The Rule Descriptor Table contains information of a rule such as the rule identification, the valid time, the trigger conditions, the pointer to the stored tree in the Rule Body Table, and the number of entries the tree occupy, etc. The Rule Body Table stores the trees of the parsed rules.

Temporal rules of a knowledge base may have conflicts and redundancies. The validation of temporal rules is essential to ensure the consistency and correctiveness of the knowledge base. As data are entered into the knowledge base, the validated temporal rules can be used to maintain the knowledge base. When a temporal rule is updated, the rule validation process need to be performed again, and the existing data will have to be evaluated against this new rule. Rule validation is a non-trivial problem and is out of the scope of this dissertation. In this work, we assume that the rule base has been validated and shall concentrate on the management of temporal knowledge rules.

4.3.1 Modeling Rules as Objects

Temporal rules are metadata or meta knowledge of a knowledge base. They are part of the semantic properties of object instances and classes and are useful to the users of the knowledge base. For instance, a query such as "why could Ritter be a senior engineer and a sales manager at the same time five years ago?" will require the processing of both the data and the rules of "five years ago" to obtain a correct answer. Therefore, it is important and necessary for a system to model and manage temporal rules so that they can be retrieved and processed just like application data for various purposes.

In our implementation of an OO object manager based on C++, temporal rules are modeled as first class objects [YAS91] similar to the approach taken in [DAY88b]. Figure 4-2 shows part of the meta model of the system. The root of the class system is the class named OBJECT which contains all the objects of a knowledge base. E-CLASS_OBJECT and D-CLASS_OBJECT are the subclasses of the class OBJECT and are used to model system-named and self-named objects, respectively. The former models all objects of interest in an application domain whose identifiers are assigned by the system. The latter models objects named by their values which are used to describe or characterize E-CLASS and/or D-CLASS objects (e.g., integer 5, character string "John," etc.) Since the modeling constructs of OSAM*/T such as temporal rules, classes, associations, and methods are treated as system-named objects, they are modeled as subclasses of E-CLASS_OBJECT and named as TEMPORAL_RULE, CLASS, ASSOCIATION, and METHOD, respectively. The class named CLASS in the figure contains all the definitions of classes in the entire system. A class is defined by a class name, a set of temporal rules, a set of associations and a set of methods. The

definitions of all the classes (entity and domain classes) shown in the figure including the class CLASS itself are instances of CLASS.

The class TEMPORAL_RULE in Figure 4-2 contains all the temporal rules used in the system as its instances. As object instances, they can be retrieved and processed as ordinary object instances. Each of these instances has a system-defined identifier (IID) and a user-defined rule identifier (RID) for unique identification when it is initially created. As illustrated in Figure 4-3, the components of a rule (e.g., RID, triggered-clause, condition-clause, action-clause, etc) are modeled as descriptive attributes of the class TEMPORAL_RULE. "RID" is a user-defined identifier, "belongs_to_class" specifies the class that a temporal rule belongs to, "triggered-clause" specifies the triggering condition of a temporal rule, "condition-clause" specifies the condition part of a temporal rule, "action-clause" and "otherwise-clause" specify the consequence part of a temporal rule. The three types of temporal rules described before (i.e. state rules, operational rules, and deductive rules) are modeled as subclasses of TEMPORAL_RULE and named as STATE_RULE, OPERATIONAL_RULE, and DEDUCTIVE_RULE, respectively.

4.3.2 Uniform Treatment of Temporal Data and Rules

The constraints of a knowledge base defined by temporal rules can become out-of-date as the knowledge base evolves. In that case, the rules become invalid and need to be updated or deleted. The updated or deleted rules become historical rules and are no longer valid to the current knowledge base. However, they are still valid to some historical data and it is important for a system to maintain the histories of temporal rules.

In OSAM*/T, we use the same object instance time-stamping technique to model and record the histories of temporal rules. Using this technique, the evolution of a temporal rule will be recorded and managed in the same way as the evolution of any ordinary application object. A temporal rule has a Start-time to indicate the time the rule becomes valid and an End-time to indicate the time before the termination of its validity. Whenever a rule evolves, a new rule is created to replace the old rule which is shifted and stored in the historical area. It is the current temporal rules which affect and constrain the current knowledge base activities such as update, delete, insert, retrieve, and user-defined operations. The historical rules, on the other hand, are needed only when the historical data are referenced. In the following, we present examples of rule update, rule retrieval and rule evaluation.

4.3.2.1 Update of temporal rules

Updates to temporal rules are carried out by update transactions in the same manner as updating data because temporal rules are modeled as ordinary objects. An update transaction operated on a temporal rule would involve rule-modification, rule-parsing, and rule-validation before it can be committed. We use Rule 00005 as an example to illustrate the update of a temporal rule. The update operation involves the modification of the condition-clause and the action-clause. The original version and the updated version of Rule 00005 are given below:

Rule 00005
Valid_T [17,20]
triggered (After InsertObject(Work_On))
condition (exist this *in* this AND (*that*Project[P# = P5],
!Work_On*Project[P# = P6])) ^
(exist this in WHEN T[4,12]
this * Work_On * Project[P# = P1])
action associate(this * Work_On * Project[P# = P6])
End

(a) An invalid rule to the current knowledge base.

Rule 00005
Valid_T [21,-]
triggered (After InsertObject(Work_On))
condition (exist this *in* this AND (*that*Project[P# = P5],
!Work_On*Project[P# = P7])) ^
(exist this in WHEN T[4,12]
this * Work_On * Project[P# = P1])
action associate(this * Work_On * Project[P# = P7])
End

(b) A valid rule to the current knowledge base.

In this example, Rule 00005 which was defined at time 17 says that the employee who is assigned to project P5 by the currently inserted Work_On instance and who participated in project P1 during the period T[4,12] should also join project P6. After the rule was defined for a while (i.e., 4 time units later), project P6 was completed and it is no longer meaningful to assign an employee to P6. Therefore, this rule was out-of-date and was updated at time 21. The updated version says that the employee who is assigned to project P5 and who participated in project P1 during the period T[4, 12] should also join project P7. After this rule is updated, the qualified employees who is assigned to project P5 will also be assigned to project P7 instead of P6.

4.3.2.2 Retrieval of temporal rules

Since temporal rules are modeled as ordinary objects, they can be retrieved in the same way as other ordinary data. For example, if a user needs to see the information of the condition-clause of Rule 00005 at time 28, this query can be expressed as follows:

Query 4-1: AT 28
CONTEXT r:Temporal_Rules [RID = 00005]
Retrieve r.condition-clause

When this query is evaluated, the system will retrieve the new condition-clause of Rule 00005 and the answer will be the following:

condition (exist this *in* this AND (*that*Project[P# = P5],
!Work_On*Project[P# = P7])) ^
(exist this in WHEN T[4, 12]
this * Work_On * Project[P# = P1])

4.3.3 Evaluation of Temporal Rules

The evaluation of temporal rules in OSAM*/T is based on the Match-Modify-Execute (MME) cycle proposed in [RAS88] and the nested transaction model proposed in [MOS85]. A first level database transaction (i.e., update, retrieve, insert, delete, etc.) is parsed into a query tree and is matched with the trigger conditions (i.e., trigger time and trigger operation) of the rules associated with the object(s) being operated on. If there is a match between the trigger operations of these rules and the operation in the transaction, then those rules would be selected for rule evaluation. Once the "match" is successful, the original transaction is "modified" to incorporate the processing of the triggered rules (i.e., both the evaluation of the "condition" and the execution of the "action"). The processing of the triggered rules then are treated as sub-transactions which are nested under the first level transaction. It is possible that a database transaction will go through several MME cycles (i.e., several layers of nested transaction) before it commits due to the continuous triggering of temporal rules (i.e., a triggered rule triggers another rule). More detailed descriptions of the implementation and evaluation of rules can be found in [RAS88, CHU90, SIN90, ARR92].

In the following, we explain the evaluation of temporal rules using the updated Rule 00005 as an example. When the operation of inserting an object instance into the Work_On class is detected by the system (i.e., the application is assigning an employee to a project), the updated Rule 00005 defined in Employee class will be triggered to evaluate the quantified expressions specified in the condition-clause. In evaluating the expressions, the system will bind "that" to the inserted Work_On instance in the

expression "this AND (* that * Project[P# = P5], ! Work_On * Project[P# = P7])". The employee who is being connected to P5 by the insertion of "that" instance is identified by the keyword "this". The condition-clause verifies if this employee instance falls in the association patterns specified. If so, the action-clause is executed which associates this employee with project P7. For example, if the instance W1, which assigns employee John to project P5, is entered into Work_On class. Assuming that John is not involved in project P7 and worked on project P1 during T[4,12], the result of evaluating the condition-clause of this rule will be True. The associate() operation in the action-clause will be executed and the result is the establishment of the association pattern "John * W11 * P7" in the knowledge base as shown in Figure 4-4.

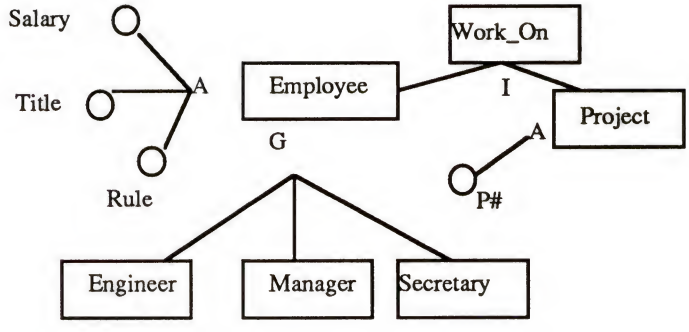


Figure 4-1: S-diagram of a company database.

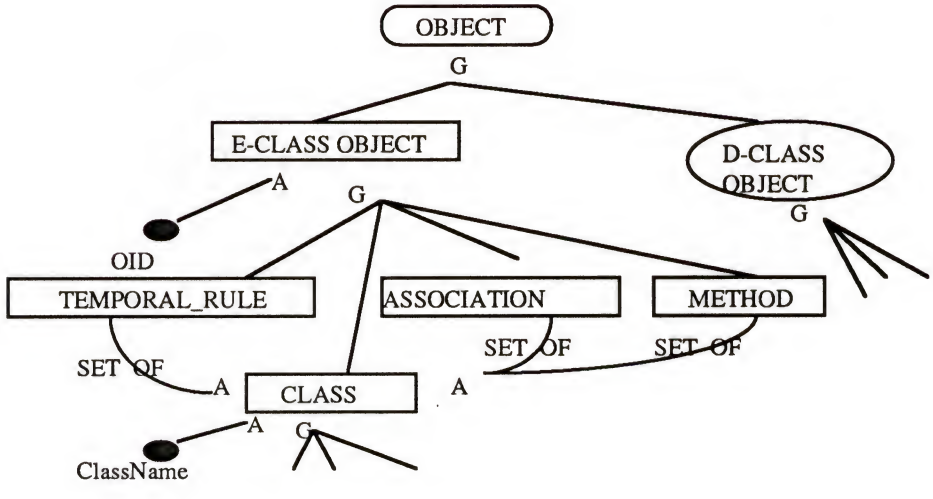


Figure 4-2: The meta model of the system.

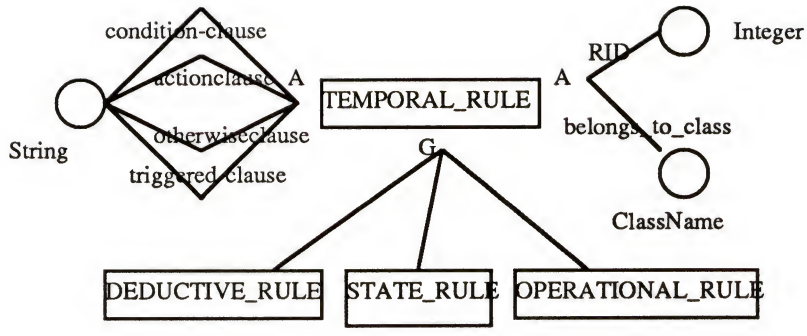


Figure 4-3: Temporal rule class hierarchy of the OSAM*/T model.

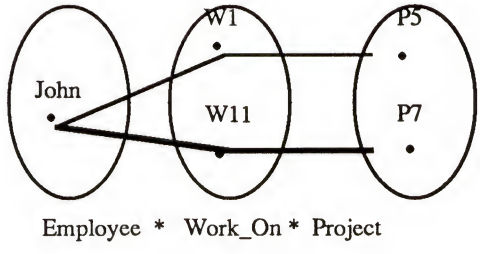


Figure 4-4: Establishment of the pattern "John*W11*p7" after rule 00005 is fired.

CHAPTER 5
THE TEMPORAL QUERY LANGUAGE: OQL/T

This chapter describes the temporal query language OQL/T which is the query language for processing temporal databases modeled by OSAM*/T. It is a super set of the object-oriented query language OQL [ALA89] with extensions of temporal constructs and functions for specifying temporal conditions. The general structure of a query in OQL/T is shown as below and the BNF is given in Appendix B:

```
WHEN temporal_conditions
CONTEXT association_pattern_expression
        WHERE conditions
        SELECT object classes and/or attributes
OPERATION(s) object class(es)
```

The main difference between the query structures of OQL/T and OQL is that the query structure of OQL/T contains an additional temporal condition (WHEN clause) which specifies the temporal snapshot databases that will be processed. The association pattern specification of OQL still plays a key role in the specification of a temporal query in OQL/T and is called temporal association pattern. In the following subsections, we shall describe and illustrate temporal association patterns, the specification of WHEN clause, the interval comparison operators which is useful for specifying temporal conditions in temporal association patterns, and the constructs and functions introduced for specifying the temporal conditions of data reference in the WHEN clause.

5.1 Temporal Association Patterns

A temporal association pattern is an intensional representation of a set of classes and their association relationships. It shows how objects in the classes specified in the expression are linked to one another during some time intervals. Different from query specification in a conventional DBMS, the association-based query formulation allows users to query a temporal OO database by simply specifying time intervals and patterns of object associations as search conditions. The advantage of this formulation for querying a database is the simplicity in specifying complex queries that involve multiple classes. Once the objects which satisfy an association pattern specification have been selected, they can be further processed by either system-defined operations (Retrieval, Display, Update, Insert, Delete, etc.) or user-defined operations (RotatePart, PurchasePart, HireFaculty, etc.) An example of association-based temporal query formulation are given below:

```
WHEN T[3,6]
CONTEXT e[1]:Employee AND (*Work_On * Project[P# = pj1],
                           *Work_On * Project[P# = pj2])
Retrieve e.Salary
```

In this example, the temporal query retrieves the salaries of those employees of the period T[3,6] who satisfy the association pattern specified in the CONTEXT statement. The pattern specification identifies those employee instances that are connected to pj1 via a Work_On instance and to pj2 via another Work_On instance. The query pattern is more clearly illustrated in Figure 5-1. All the employees of the period T[3,6] who satisfy the pattern will be retrieved. This query illustrates a tree structured query having an

---

[1]The expression "e:Employee" is an assignment of the Employee class to the instance variable e as explained in Chapter 4. Any occurrence of e after this assignment in this query refers to the set of employee instances satisfying the condition(s) as stated in the context-clause.

AND branch. Complex queries with AND/OR branches and loops can also be specified by patterns with relative ease. We note here that if there are multiple associations between two classes, the association name of a specific association link should be specified right after the '*' operator in a query.

5.2 The WHEN Clause

The WHEN clause in OQL/T is optional and is used to specify the interested snapshots of the temporal database. It consists of the keyword WHEN followed by a temporal_condition specifying the time interval of interest. If WHEN clause does not appear in an OQL/T query, the current database is assumed in the processing of the query.

The specification of time information in the WHEN clause can be classified into two types: time and data references. In the first type, the keyword WHEN is followed by an explicit time interval of the form "T[A,B]," where A and B are time points and A is less than or equal to B. If A is equal to B, the time reference is called time-point reference and the "WHEN T[A,B]" can also be replaced by "AT A"; if A is less than B, the time reference is called a time-interval reference. In both cases, the query will be evaluated against the temporal snapshot databases between A and B. Examples 5-1 and 5-2 illustrate the uses of time-interval and time-point references, respectively.

Example 5-1: How many times has Mary's Title been changed during the period between time 3 and time 6?

```
WHEN T[3, 6]
CONTEXT e:Employee [Name = "Mary"]
RETRIEVE TCOUNT(e.Title)
```

Example 5-2: What is John's Title at the time 11?

```
WHEN T[11, 11]
CONTEXT e:Employee [Name = "John"]
RETRIEVE e.Title
```

or

```
AT 11
CONTEXT e:Employee [Name = John]
RETRIEVE e.Title
```

If the specification of temporal condition is data reference, WHEN is followed by an interval expression of the form "INTERVAL (Data Condition)". In general, the Data Condition can be a simple object instance or a complex association pattern expression containing interval comparison operators, temporal ordering functions, and/or a set of temporal functions such as NEXT, FORMER, NOW, and etc. In any case, a query will be evaluated against those temporal snapshot databases which satisfy the Data Condition. Examples 5-3 and 5-4 illustrate the uses of Data Conditions in a simple and a complex case, respectively.

Example 5-3: What is Mary's salary when she was a clerk?

```
WHEN INTERVAL(e:Employee [Name = "Mary", Title = "Clerk"])
CONTEXT e
RETRIEVE e.Salary
```

Example 5-4: What was Mary's salary when John worked on Project P1?

```
WHEN INTERVAL (Employee[Name="John"] * Work_On * Project[P#=P1])
CONTEXT e:Employee [Name = Mary]
RETRIEVE e.Salary
```

In Example 5-3, Employee in the CONTEXT clause is bound to the Employee specified in the WHEN clause (i.e., the employee Mary). In Example 5-4 the association operators "*'s" specify that the employee John is associated with an object instance of Work_On class which is associated with project P1. The data condition of this query specifies the time interval desired in the query.

A data condition in a WHEN clause sometimes will return multiple intervals. When multiple intervals are returned, different strategies may be used. One strategy is to evaluate the query against all these intervals; the other is to select only one interval for query evaluation. In OSAM*/T, we choose to evaluate a query against all qualified

intervals. When there is a bind between a class in the data condition and a class in the CONTEXT clause, the query will be evaluated iteratively for each bound object instance of the class; otherwise, if there is no relationship between the classes of the WHEN and CONTEXT clauses, the query will be evaluated in a set-oriented fashion.

Since the use of time reference in a OQL/T query is straight forward, we shall emphasize on the specification of data reference and elaborate on the possible Data Conditions by examples. In the following sections, we introduce the proposed temporal constructs and functions which can be used to express various Data Conditions. Most of the examples given in these sections are based on the database given in Table 5-1 and Table 5-2 which contain employee Nancy's and Joe's histories, respectively (the employee instances Nancy and Joe are modeled by the IIDs 051 and 052, respectively).

5.3 Interval Comparison Operators and Functions

Because we use the Start-time and End-time tags of valid time notion to uniquely characterize the temporal object instances, the information of the interval between the two time points is relevant to temporal data manipulation [ALL84, TAN86, DE87, AHN88, LOR88]. In this dissertation, we refer a time interval to a pair of time points and the duration between them. A time interval is expressed as "$T[t_1, t_2]$," where $t_1 <= t_2$, and the special symbol T[] is reserved for expressing the time interval which contains the Start-time and End-time.

A time interval can be assigned to a variable. For instance, we can assign the above time interval (i.e. the time points t_1 and t_2) to variable "A" as "$A = T[t_1, t_2]$". When a time interval is assigned to a variable, that variable is equivalent to the interval it represents and can be used to express temporal conditions in a query.

The interval specified in the WHEN clause may subject to some temporal conditions. In this case, the WHERE subclause is used to specify a Boolean expression of time intervals and interval comparison operators. The expression specifies how the interval following the WHEN clause is related to some other time intervals. If the expression in the WHERE subclause is evaluated to True, the rest of the query will be processed against the snapshot database defined by the interval following WHEN. Otherwise, the query will not be executed. In the following, we discuss all the possible relationships between two time intervals and identify those which are useful to the specification of temporal queries. Each of the identified relationship between two time intervals is represented by a keyword as an Interval Comparison Operator.

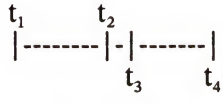
Given two time intervals $A=T[t_1,t_2]$ and $B=T[t_3,t_4]$, where $t_1 \leq t_2$, and $t_3 \leq t_4$, we identify systematically the following possible temporal relationships modeled by sixteen interval comparison operators.

Case (I): $t_1 = t_2$

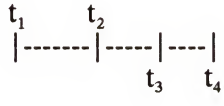
(1) $t_3 = t_4$

(i) $t_2 < t_3$

(a) $t_3 - t_2$ = time unit, A PRECEDING B, or B FOLLOWING A



(b) $t_3 - t_2 >$ time unit, A BEFORE B, or B AFTER A



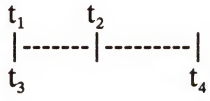
(ii) $t_1 = t_3$, A EQUAL B

(iii) if $t_1 > t_4$, similar to $t_2 < t_3$ in I.1.i

(2) $t_3 < t_4$

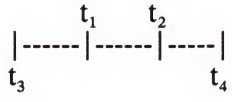
(i) $t_2 < t_3$, similar to $t_2 < t_3$ in I.1.i

(ii) $t_1 = t_3$, A L-WITHIN B, or B L-CONTAIN A (L stands for Left)



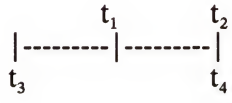
(iii) $t_3 < t_1 < t_4$, A I-WITHIN B, or B O-CONTAIN A

(I stands for Inner and O stands for Outer)



(iv) $t_3 < t_1 = t_4$, A R-WITHIN B, or B R-CONTAIN A

(R stands for Right)



(v) $t_1 > t_4$, similar to $t_2 < t_3$ in I.1.i

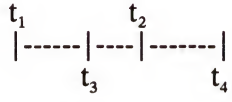
Case (II): $t_1 < t_2$

(1) $t_3 = t_4$, similar to the case of I.2

(2) $t_3 < t_4$

(i) $t_2 < t_3$, similar to $t_2 < t_3$ in I.1.i

(ii) $t_1 < t_3 <= t_2 < t_4$, A P-CROSS B, or B F-CROSS A



(iii) $t_1 = t_3 < t_2 < t_4$, similar to I.2.ii

(iv) $t_1 < t_3 < t_4 < t_2$, similar to I.2.iii

(v) $t_1 < t_3 < t_2 = t_4$, similar to I.2.iv

(vi) $t_1 = t_3 < t_2 = t_4$, similar to I.1.ii

In addition, we also use CROSS to imply any case of P-CROSS and F-CROSS; CONTAIN to imply any case of O-CONTAIN, L-CONTAIN, and R-CONTAIN; WITHIN to imply any case of I-WITHIN, L-WITHIN, and R-WITHIN.

5.4 Temporal Functions

5.4.1 INTERVAL

INTERVAL is a function which takes a temporal predicate as the input and produces a time interval as the output. A temporal predicate is defined as a predicate with temporal constraint. For example, "Nancy was a secretary during time 12 to 17" is a temporal predicate. The duration T[12,17] when Nancy was a secretary can be retrieved by the INTERVAL function and a temporal predicate expressed as:

INTERVAL (Employee[Title = "Secretary" ^ Name = "Nancy"])

An example of the use of the INTERVAL function is to retrieve Joe's salary when Nancy was a secretary. This query is given in Example 5-5 and is expressed as below:

Example 5-5: What was John's salary when Nancy was a secretary?

```
WHEN INTERVAL(Employee [Name ="Nancy", Title ="Secretary"])
CONTEXT e:Employee[Name = "Joe"]
RETRIEVE e.Salary
```

In this query, the Employee [Name = "Nancy", Title = "Secretary"] in the WHEN clause is used only as the data condition to specify the time interval that the rest of this query is to be evaluated upon. Employee in the CONTEXT clause is another scan of the Employee class and is used only to specify the pattern of object instances of interest. The interested pattern of object instances are assigned to the instance variable e. The instances of Joe's history which have a WITHIN, CROSS, or CONTAIN relationship with the interval when Nancy was a secretary are the qualified instances, and will be used as the context for answering this query. In this example, Joe's salaries of $42K

during T[9,14] and $45K during T[15,-] will be retrieved because T[9,14] and T[15,-] both have a CROSS relationship with T[12,17] (see Table 5-1 and 5-2).

5.4.2 FORMER

FORMER is a function used to retrieve the former historical instance (or event) relative to a referred instance specified in a temporal predicate. The general form of FORMER can be expressed as below:

FORMER (parameter) --> historical object instance

where, parameter is a temporal predicate and the output is the historical instance happened prior to the instance in the predicate. An example of this function is to find all the employees who had been "Senior Engineer" right before they were promoted to "Project Supervisor" as shown in Example 5-6.

> Example 5-6: Find the employees who had been "Senior Engineer" right before they were "P. Supervisor" ?
>
> ```
> WHEN INTERVAL(FORMER (e:Employee[Title="P. Supervisor"]))
> CONTEXT e [Title = "Sr. Engineer"]
> RETRIEVE e.Name
> ```

In this example, the object instances of Employee in the WHEN clause is used to specify the time interval and to serve as an anchor for the object instances of the Employee class in the CONTEXT clause. Therefore, an assignment of the Employee class to the instance variable e is specified in this query. When the query is evaluated, the system will first search for all the object instances in the Employee class to check if any of their historical instances has a title Project Supervisor. Then, based on this reference, the system will check if the former title relative to Project Supervisor is Senior Engineer or not. For those qualified object instances, their names will be retrieved. In our database example (Table 1 and 2), only Joe ever was a Project Supervisor and his

former title is Engineer instead of Senior Engineer. Therefore, there is no object instance which satisfies this temporal condition and the result of this query is null.

5.4.3 NEXT

NEXT is a function used to retrieve the historical instance that follows a reference instance. It is symmetric to the FORMER function. The general form of NEXT can be expressed as "NEXT (parameter) --> historical instance".

5.4.4 TIME:

TIME is a special function used to return a time point. The parameter of this function can be NOW, which is a keyword representing the current time. The "+" and "-" symbols are used together with NOW to indicate the relative time to the current time. For example, "NOW + 3" stands for three time units ahead of the current time; and "NOW - 2" stands for two time units behind the current time. The general form of this function is shown below:

TIME (NOW +/- x time units) -->

x time units ahead/behind the current time, where x is an integer.

The meaning of TIME(NOW +/- x), therefore, is the projection of the relative time point to current time when this function is evaluated. When x is not given, TIME(NOW) is a projection of the current time.

5.4.5 START & END

START and END are the two functions in OQL/T used to retrieve time tags of historical versions of object instances. START is used to retrieve the start time and END is used to retrieve the end time of a historical object instance. When they appear

in a query, the start time and/or the end time of a historical object instance will be returned. An example of retrieving the time when Mary became a manager is shown in Example 5-7:

Example 5-7: When did Nancy become a manager?

```
WHEN INTERVAL(e:Employee[Name="Nancy" ^ Title="Manager"])
CONTEXT e
RETRIEVE START(e)
```

In this example, the system will search through Nancy's record for the historical instances which have the title Manager. Once the instance is found, the start time of the interval during which Nancy's title is Manager will be the answer. Therefore, 25 is the answer to this query according to Table 5-1. The use of END is the same as that of START except the end time of an instance will be returned instead.

5.5 Temporal Ordering Functions

The main concept of temporal ordering of an object instance's history is to sort the historical versions of an object instance in an ascending order based on their time stamps so that retrieval of historical records of a specified order is possible [NAV89]. In OSAM*/T, we introduced FIRST, LAST, and NTH as the forward temporal ordering functions and B_FIRST, B_LAST, and B_NTH (where B stands for backward) as the backward temporal ordering functions. Forward temporal ordering functions are used to retrieve object instances in a forward order; whereas the backward temporal ordering functions are used to retrieve object instances in a backward order. The parameter for functions of FIRST, LAST, B_FIRST, or B_LAST is an object instance or a historical event, and the output is either the first object instance or the last object instance depending on the function used. The parameters for functions NTH and B_NTH are a number and an object instance, and the output is the object's historical record of the

order specified by the number. Example 5-8 and 5-9 illustrate the use of forward and backward temporal ordering functions, respectively.

> Example 5-8: Retrieve the names and salaries of those employees whose starting salaries were greater than $25K.
>
> WHEN INTERVAL (FIRST(e:Employee))
> CONTEXT e [Salary > $25K]
> RETRIEVE e.Name, e.Salary
>
> Example 5-9:Retrieve the names and salaries of the last second record of those employees whose salaries were greater than $37K.
>
> WHEN INTERVAL (B_NTH(2,e:Employee))
> CONTEXT e [Salary > $37K]
> RETRIEVE e.Name, e.Salary

In Example 5-8, the system will compare the salary of the first record of each temporal object instance with $25K. If the condition is satisfied, the employee's name and salary will be retrieved. In our database example, only Joe's first historical instance (see Table 5-2) satisfies the condition and will be retrieved. In Example 5-9, the system will compare the salary of the second record in the backward order of each temporal object instance with $37K. Joe's historical record during T[9,14] satisfies the condition and will be retrieved.

5.6 Moving Window Functions

The concept of "Moving-Window" [NAV89] is introduced for some statistics applications. A Moving-Window is a period of time which moves at a constant pace from the lower bound toward the upper bound of a wide range interval. In a "Moving-Window" application, conditions in a query will be evaluated as many times as the period shifts from the lower bound toward the upper bound of the range interval at a specified constant pace. That is, each time the period shifts, the conditions in a query will be re-evaluated in the new specified period.

For the convenience of specifying the concept of Moving-Window efficiently, we introduce another two constructs ANY and EVERY in OQL/T. When ANY and EVERY are used as the Moving-Window, they are followed by a period of time C, a valid temporal range T[A, B], and the periodical pace D. The general form for ANY and EVERY is shown as below.

```
WHEN INTERVAL(parameter) ANY/EVERY C
  WITHIN T[A, B] INCREMENT_BY D,
```

where, parameter can be a temporal predicate; A and B are constants which indicate the lower and upper bounds of the wider temporal range; C is a constant period which is the duration of the moving time period; and D is the pace that the moving time period C is supposed to advance after each evaluation of conditions specified in the query.

Conditions specified in a query is to be evaluated upon the snapshot temporal databases within period C. The lower bound and upper bound of the period C must be WITHIN T[A,B]. When the evaluation of the conditions is finished with the first period C, the lower bound and upper bound of C advance at the pace D to form the second period C. The operation in the query will then be evaluated upon the snapshot temporal databases of the second period C. This process continues until C is no longer WITHIN T[A, B].

If D is not given, the period C will advance at the pace of the time unit specified in C. For example, if C is "3 years" then the advance pace is one year; if C is "36 months" the advance step is one month and so on. If T[A, B] is not given, it will be defaulted to the lifespan of the historical object instance in the parameter. If an object instance's history starts later than A, then the lower bound of the first period C will be the starting time of the first instance of that object.

ANY, which is similar to the term "Moving-Window" in [NAV89], is used to capture "there exist" concept; whereas EVERY is used to capture "for all" concept.

Furthermore, the general forms of ANY and EVERY also allow a user to specify any moving pace of a moving window and make default assumptions for the interval T[A, B] and the moving pace D if they are not specified. Examples for their usages are shown in Example 5-10 and 5-11.

Example 5-10: Find the employees whose titles had been changed more than twice within any 9-time-unit period during time points 6 and 18.

```
WHEN INTERVAL(e:Employee) ANY 9-time-unit
    WITHIN T[6, 18] INCREMENT_BY 1-time-unit
CONTEXT e
        WHERE TCOUNT(e.Title) > 2
RETRIEVE e.Name
```

Example 5-11: Find the employees whose titles had been changed more than twice within every 9-time-unit period during time points 6 to 18.

```
WHEN INTERVAL(e:Employee) EVERY 9-time-unit
    WITHIN T[6,18] INCREMENT_BY 1-time-unit
CONTEXT e
        WHERE TCOUNT (e.Title) > 2
RETRIEVE e.Name
```

In Example 5-10, a 9-time-unit period will be applied to the interval T[6,18]. Both the lower bound and the upper bound of this 9-time-unit period are limited to be within T[6,18]. In our database example, the lower bounds of the first 9-time-unit period for object instances Nancy's and Joe's history are both bound to 6. Condition "TCOUNT (Title) > 2" in the WHERE subclause is evaluated over the first 9-time-unit period. If the evaluation does not produce result, the second 9-time-unit period is formed by advancing the first 9-time-unit period by 1-time-unit as specified in the query and the condition will be re-evaluated. This process continues until either the condition is satisfied or the upper bound of the 9-time-unit period has been exceeded. In our database example, both Nancy's and Joe's historical record meets the specified condition. So the result for this query is Nancy and Joe.

In Example 5-11, the names of those employees whose titles had been changed at least twice in every 9-time-unit period within T[6,18] will be retrieved. When this query is evaluated, the history of every object instance will be tested over the snapshot temporal databases of all the possible 9-time-unit periods. In our database example, no employee instance satisfies the temporal conditions in this query.

5.7 Set Operators

In some applications, it is necessary to involve the temporal information of different snapshot databases for the data manipulation of a particular snapshot database. The temporal information in these applications are used as the restricting condition of the time dimension on the interested temporal object instances. In order to achieve this cross-time referencing, we introduce the Set Operators such as NT-INTERSECT, INTERSECT, DIFFERENCE, and UNION.

The syntax of a query that involves a Set Operator is given as following:

```
WHEN temporal-condition-1
CONTEXT association-pattern-expression-1
            WHERE condition-1
Set-Operator (Target-Classes)
WHEN temporal-condition-2
CONTEXT association-pattern-expression-2
            WHERE condition-2
```

The operands of a Set Operator are two temporal contexts which define two separate temporal subdatabases. The result of the set operation is a subdatabase derived from the two subdatabases. One restriction on the two contexts is that there must be at least one intersecting class between them and the operation of the Set Operator is performed on the intersecting class(es). For example, Set Operators can not be applied to the two contexts A*B*C and D*E*F because there is no intersecting class between them; however, Set Operators can be applied on the two contexts A*B*C and A*B*D*E

because there are two intersecting classes A and B. In the later case of the above example, a set operation can be applied on class A, class B, or classes A and B depending on the user's requirement. The "Target Classes" following the Set Operator is used to specify the intersecting classes on which the Set Operator is performed. "Target Classes" is member of the power set of all intersecting classes. If it is not provided, the member with a maximal number of intersecting classes from the power set will be used as the default. The mathematical properties of the set operators and their implementation strategies are formally defined in Chapter 6.

An OQL/T query can take advantage of the cross temporal referencing capability of a Set Operator to capture complex requirement. In general, an OQL/T query can contain multiple contexts with each context linked to another through a Set Operator.

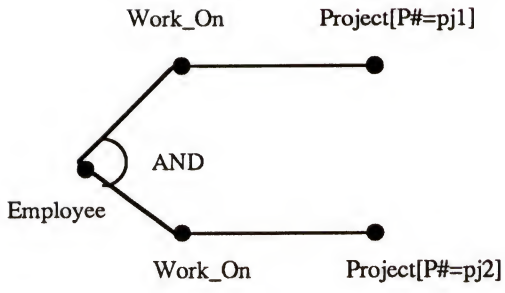


Figure 5-1: Illustration of the query pattern that an employee works on both pj1 and pj2.

Table 5-1: Employee instance Nancy's history.

Start	End	IID	Title	Dept	Salary	
<25,	-,	051,	Manager	SALES	$40K>	. . . . . current version
<18,	24,	051,	Supervisor	SALES	$33K>	. . . . . 4th version
<12,	17,	051,	Secretary	SALES	$27K>	. . . . . 3rd version
< 8,	11,	051,	Clerk	SALES	$22K>	. . . . . 2nd version
< 3,	7,	051,	Clerk	SALES	$15K>	. . . . . 1st version

Table 5-2: Employee instance Joe's history.

Start	End	IID	Title	Dept	Salary	
<15,	-,	052,	P. Manager	R&D	$45K>	. . . . . current version
< 9,	14,	052,	P.Supervisor	R&D	$42K>	. . . . . 3rd version
< 6,	8,	052,	Engr	R&D	$38K>	. . . . . 2nd version
< 1,	5,	052,	Jr Engr	R&D	$27K>	. . . . . 1st version

CHAPTER 6
TEMPORAL ASSOCIATION ALGEBRA: A MATHEMATICAL FOUNDATION FOR OBJECT-ORIENTED TEMPORAL KNOWLEDGE BASES

We have described an object-oriented knowledge base management approach to modeling and processing temporal knowledge base including the OSAM*/T knowledge model in Chapter 3, the specification and management of temporal knowledge rules in Chapter 4 and the OQL/T high-level query language in Chapter 5. For supporting high-level temporal data models and temporal query languages, it is important to identify a set of primitive algebraic operators and their mathematical properties and use them to implement high-level language constructs and to optimize queries. Mckenzie & Snodgrass [McK91] have also pointed out that "implementation issues such as query optimization and physical storage strategies can best be addressed in terms of the algebra" and that "one of the reason for the success of the relational model is that it lends itself to an algebraic execution paradigm". Although there are numerous recent efforts [SU91, ROS91, WUU92, SHI81, BAN89, BAN88, ROW87, CAR88, DAD86, COL89, FIS87, ALA89] in object-oriented temporal databases and query languages, a temporal algebra, which provides the mathematical foundation for processing object-oriented, temporal databases is still lacking.

This chapter presents a temporal association algebra called TA-algebra as a mathematical foundation for implementing object-oriented temporal knowledge bases. TA-algebra provides a set of primitive algebraic operators for manipulating temporal information of a temporal object-oriented knowledge base. It can be used to implement

the concepts and constructs proposed in the high-level model OSAM*/T and query language OQL/T described in the previous chapters. In TA-algebra, we extend the nine operators of the A-algebra [GUO91, SU93] with snapshot semantics [GAD86, GAD88, CLI85, CLI87, JEN92] resulting in the following operators: T-Associate(*), TA-Complement(|), TA-Select(δ), TA-Project(π), T-NonAssociate(!), TA-Intersect(•), TA-UNION(+), TA-Difference(-) and TA-Divide(÷). In addition, we introduce an additional operator NT-Intersect(○) which intersects subpatterns without regard to their valid times and is a useful operation for retrieving temporal data when a cross-time reference is necessary. Analogous to the A-algebra, TA-algebra is association-based, i.e., the domain of the algebra is sets of temporal association patterns (e.g., three dimensional linear structures, trees, lattices, network, etc.) and processing a temporal OO database is based on the matching and manipulation of both homogeneous and heterogeneous temporal patterns of object associations. The closure property of A-algebra is retained in TA-algebra to allow it to operate on a set of homogeneous or heterogeneous temporal patterns of object associations and to return a set of homogeneous or heterogeneous temporal patterns of the same type.

This chapter is organized as follows. We first illustrate the concepts of intensional as well as extensional temporal databases. We then formally define the concepts of Schema Graph (SG), Temporal Object Graph (TOG), temporal association pattern instance, snapshot of temporal association pattern instance, temporal association pattern set, and snapshot of temporal association pattern set. We also introduce five primitive temporal patterns and the concept of temporal states which are useful for decomposing a temporal association pattern instance into primitive patterns for uniform treatment by the TA-algebra operators. Definitions of temporal association operators of TA-algebra and their mathematical properties ensue. Furthermore, various methods for

specifying time intervals in TA-algebra are described and the algebraic expressions for some example queries are given to demonstrate the utility of the algebra. Lastly, we compare the functionality and expressive power of TA-algebra with some related works.

6.1 Intensional vs Extensional Temporal Database

From the algebra point of view, a temporal OO database can be viewed as a collection of objects and their historical versions grouped together in classes and interrelated through associations along the time dimension. It can be represented by graphs at both the intensional and the extensional levels. At the intensional (schema) level, a temporal database is defined by a collection of inter-related object classes and is represented by a Schema Graph (SG). For example, the SG for a simple company database is illustrated in Figure 6-1, in which each rectangle denotes an entity-class such as a class of person objects or a class of department objects, and each circle denotes a domain-class such as a class of names or salaries. The associations among classes are represented by the edges in SG. The label on one end of each edge denotes an association type such as A for aggregation, G for generalization and I for interaction. The schema models the Work_On relationship between Employee and Project, and the superclass-subclass (or generalization) association between Person and Employee.

At the extensional (instance) level, a database can be viewed as a collection of object instances grouped together in classes and inter-related through some object associations; each object instance in a class is associated with a set of historical versions resulting from its evolution. As such, an extensional temporal database can be represented by a Temporal Object Graph (TOG) which is a set of current and historical object instances connected by version links and object associations into a three dimensional network structure. For example, the TOG for a portion of the company

schema of Figure 6-1 is shown in Figure 6-2 which contains the temporal information of a particular person (modeled as p1 in Person class) who is an employee (modeled as e1 in Employee class) working on certain projects (associated by the instance w1 of the Work_On class). The historical versions of an object instance are linked by version pointers in TOG in the manner that a latter version points to its previous version. For example, the historical versions of p1, e1 and w1 are linked by version pointers from current versions toward their earlier versions. Each version of an object instance in TOG is connected to its associated object instances by solid_line arrows (—►) pointing from the historical version (identified by a TIID) to the IIDs of the associated object instances (which are represented by dotted circles around the versions of temporal object instances). The object associations are bidirectional (i.e., object instance traversals can go both ways); however, we use solid_line arrows (instead of just regular edges) in TOG to clearly indicate that a historical version of an object instance is defined in terms of its association with an object instance pointed to by the arrow. For example, in Figure 6-2, the second version of p1 which prevails the time interval T[5,7] has pointers pointing to the identifiers of a2 in Address and e1 in Employee, respectively, to represent the fact that p1 is defined by its associations with a2 and e1 during this time interval. The versions included in the time interval T[5,7] of all the pointed object instances also have pointers pointing to the identifier of p1 because they are associated with p1 during this time interval.

6.2 Temporal Association Algebra

The Temporal Association Algebra (or TA-algebra) consists of ten operators for processing temporal data in the time dimension based on the snapshot semantics. These operators operate on graph structures of temporal object associations to produce graph

structures. The closure property of TA-algebra ensures that the result of a temporal algebraic operation can be further manipulated by other operators.

In this section, we formally define the following temporal concepts which will be used in the explanation of TA-algebra operators: Schema Graph (SG), Temporal Object Graph (TOG), temporal association pattern instance, snapshot of temporal association pattern instance, temporal association pattern set, snapshot of temporal association pattern set, homogeneous temporal association pattern set, and heterogeneous temporal association pattern set.

6.2.1 Schema Graph: the Intensional Temporal Database

The schema graph of a temporal OO database is defined as SG(C,A), where $C=\{C_i\}$ is a set of vertices representing temporal object classes each of which has a time interval [CLI87, GAD88, ROS92] during which the class is valid; A is a set of edges, each of which, $A_{i,j}(k)$, represents association between classes C_i and C_j, where k is a number for distinguishing the edges from one another when there is more than one edge between two vertices. Like a class in C, each edge $A_{i,j}(k)$ has a time interval during which the edge (or association) is valid.

6.2.2 Temporal Object Graph: the Extensional Database

The temporal object graph of a temporal OO database is defined as TOG(O, E), where $O = \{O_{i,j,v}\}$ is a set of vertices (or TIIDs) representing versions of temporal object instances (the vth version of the jth temporal object instance in class C_i); E is a set of solid_line arrows (—►) representing association pointers between historical versions of temporal object instances and the associated temporal object instances. Each vertex in TOG is associated with a valid time interval representing the validity of a particular

version of a temporal object instance. Temporal object instances of a domain class are valid when the domain class is defined and are associated with a valid time interval T[1,NOW]. The versions (or vertices) of a temporal object instance are linked together through version pointers in a backward time order, i.e., from the most recent version toward the earliest version. In a temporal database, when two temporal object instances $O_{i,j}$ and $O_{m,n}$ are associated with each other during some time period, we draw association pointers from all the historical versions (or TIIDs) v of $O_{i,j}$ of these time period toward the identifier of $O_{m,n}$ (which is the IID of $O_{m,n}$ and is represented by a dotted circle around its historical versions) as $O_{i,j,v} \longrightarrow\!\blacktriangleright^{k} O_{m,n}$ and from all the historical versions (or TIIDs) u of $O_{m,n}$ of these time period toward the identifier of $O_{i,j}$ as $O_{m,n,u} \longrightarrow\!\blacktriangleright^{k} O_{i,j}$, where k represents the kth association between classes C_i and C_j. The reason for pointing from a version of temporal instance to an object instance instead of to a specific version of an instance is that time interval specifications for the temporal instances of different classes can be different and the data of a pointed-to instance have to be determined based on the time interval of the temporal instance pointing to it. The object associations are bidirectional (i.e., they can be traversed in both directions). However, we use solid_line arrows (instead of just regular edges) in TOG to clearly indicate that a particular historical version of an object instance is associated with some object instances. For example, in Figure 6-2, association pointers are drawn from all the historical versions of p1 toward e1 and those of e1 toward p1 because p1 and e1 are associated with each other since time 1. If two temporal object instances $O_{i,j}$ and $O_{m,n}$ are not associated with each other during some time period, complement pointers are drawn from the corresponding versions of $O_{i,j}$ to $O_{m,n}$ and from the corresponding versions of $O_{m,n}$ to $O_{i,j}$ in the same manner as association pointers are drawn. A complement pointer denoted by $O_{i,j,v} \cdots\!\blacktriangleright^{k} O_{m,n}$ represents that $O_{i,j}$ and $O_{m,n}$ are not

associated with each other during the time interval that marks the validity of the vth version of $O_{i,j}$. In an object-oriented database, these non-association pointers are not explicitly stored. However, during query processing, they are explicitly represented in TA-algebra to facilitate the processing of T-NonAssociate and TA-Complement operators to be discussed in Section 3.3. For example, the non-association relationship between e1 and w1 during T[4,5] in Figure 6-2 can be drawn as in Figure 6-3.

6.2.3 Temporal Association Pattern Instance and Temporal States

A connected subgraph of a TOG is a Temporal Association Pattern Instance (TAPI) and is called alternatively a temporal pattern for short. A TAPI α_i can be either a simple pattern which consists of a single version of a temporal object instance or a complex pattern which consists of multiple versions of many temporal object instances interconnected by association and/or non-association links into a three dimensional network structure. For example, the TAPI in Figure 6-4 (which is an abstract form of Figure 6-2) is a complex pattern with many historical versions of e1 and p1 linked by temporal associations. Each TAPI can be perceived as consisting of many temporal states each of which holds some constant object values for a period of time[1]. The prevailing period for each temporal state of a TAPI can be derived from and is conceptually similar to the valid time intervals associated with the vertices of the TAPI. Although the concept of temporal states of a TAPI is analogous to the concept of "state" proposed in [CLI83] for a tuple and the concept of "temporal structure" proposed in [TUZ90] for a relation, it is a generalization of these concepts because it is defined with respect to a TAPI which may involve many classes rather than just a single class (or

[1]That is, the object values of a temporal pattern at each time point are identical during the period associated with one temporal state.

relation). Therefore, in our work, temporal states of a TAPI are relative to the content of the TAPI: the more classes and associations that are involved in a TAPI, the more temporal states there will be for this pattern due to the asynchrounous evolution of each of its component vertex. In general, a temporal state of a TAPI transits to another state if an event such as update, insert, delete, associate, and etc. has occurred to the component vertices of the TAPI. So, if a TAPI involves only a single object instance, the temporal states of this pattern are identical to the historical versions of the object instances. We redraw Figure 6-4 by associating a time axis with each object instance in each class in Figure 6-5 to illustrate the asynchrounous evolution of object instances more clearly: p1 in Person evolved at time 5 and 8 and e1 in Employee evolved at time 4 and 6. We then use Figure 6-5 as an example to illustrate the formation of temporal states with respect to TAPIs of one class and of multiple classes. In Figure 6-5, if we are interested in a TAPI which involves only person p1 between time 1 to 6, there will be two temporal states for this pattern since p1 evolved from one historical version to another during T[1,6]: temporal state 1 of p1 which prevails T[1,4] and transits to temporal state 2 due to an address change from a1 to a2 and temporal state 2 of p1 which prevails T[5,6] (see Figure 6-6). However, if we are interested in a TAPI that involves p1, e1 and their associations between time 1 to 6 as shown in Figure 6-5, there will be four temporal states for this pattern because of the participation of e1 and the association between p1 and e1. AS shown in Figure 6-7, temporal state 1 prevails T[1,3] and transits to temporal state 2 because of the dis-associate operation between e1 and w1, temporal state 2 prevails T[4,4] and transits to temporal state 3 because of an address change of p1 from a1 to a2, and temporal state 3 prevails T[5,5] and transits to temporal state 4 because of an associate operation between e1 and w1 and a salary update of e1 from s1 to s2. Perceiving a TAPI as consisting of several temporal states is

a useful concept in query processing because the results of evaluating a query against a (set of) TAPI(s) at each time point within the period of a temporal state are identical and can simply be replaced by the result of evaluating the query only once against the temporal state of the (set of) TAPI(s). This concept is also useful for decomposing a complex TAPI into the primitive temporal patterns discussed below.

In order to introduce an algebra for processing TAPIs, these TAPIs need to be converted to set-oriented representations. In TA-algebra, we introduce five primitive association patterns and their algebraic representations so that a complex TAPI can be decomposed into a set of primitive patterns and their corresponding algebraic representations for processing by algebraic operators. They are temporal Inner-pattern, temporal Inter-pattern, temporal Complement-pattern, temporal Derived-Inter-pattern and temporal Derived-Complement-pattern. A temporal Inner-pattern is a single vertex in TOG representing one historical version of a temporal object instance. It is denoted algebraically by (start_time, end-time)a_i, where a_i is an IID, start-time and end-time define the valid time interval of the historical version of a_i. A temporal Inter-pattern captures an association in TOG between a_i and b_j. It is denoted algebraically by (start_time, end-time)a_ib_j, where start-time and end-time constitute the valid time interval of the association between a_i and b_j and can be derived from the valid time intervals associated with a_i and b_j. A temporal Complement-pattern captures a non-association in TOG between two temporal Inner-patterns a_i and b_j and is denoted by (start-time, end-time)$C(a_ib_j)$ which represents the fact that a_i and b_j are not associated with each other during the specified time interval. A temporal Derived-Inter-pattern specifies the association of two non-adjacent vertices a_i and c_j which are connected through a path of temporal Inter-patterns. It is denoted by (start-time, end-time)$D(a_ic_j)$. A temporal Derived-Complement-pattern denoted by (start-time, end-time)$DC(a_ic_j)$

specifies the non-association between two non-adjacent vertices a_i and c_j which are connected through a path containing at least one Complement-pattern. The Derived-Inter-pattern and Derived-Complement-pattern maintain the association and non-association of pairs of non-adjacent temporal object instances when the instances and associations along the path connecting them have been "projected" out.

We note here that when two TAPIs which are simply Inner-patterns are to be constructed into a TAPI of more complex pattern based on association or non-association, each of them will be considered as a subpattern of the constructed TAPI and will be eliminated in the set of primitive representation of this TAPI. For example, the two TAPIs $\{(t,t)a_i\}$ and $\{(t,t)b_j\}$ can be constructed into the Inter-pattern $\{(t,t)a_ib_j\}$ by the T-Associate operator (which will be discussed in section 3) if an association between a_i and b_j exists at time t. Based on the semantics of T-Associate operator, the result of this operation will be the set of primitive patterns $\{(t,t)a_i, (t,t)b_j, (t,t)a_ib_j\}$ which is unioned from $\{(t,t)a_i\}$, $\{(t,t)b_j\}$, and $\{(t,t)a_ib_j\}$. However, since $(t,t)a_i$ and $(t,t)b_j$ are subpatterns of $(t,t)a_ib_j$, they both will be eliminated and the result will be $\{(t,t)a_ib_j\}$.

Using the above primitives, the temporal states associated with a TAPI can be decomposed into a set of these primitives which becomes the algebraic representation of the TAPI. Each temporal primitive pattern in the set has a time interval associated with it and all the primitive patterns with the same time interval in the set constitute a temporal state. For example, the TAPI involving p1, e1 and their associations has four temporal states as illustrated in Figure 6-7. The decomposition of this TAPI can be achieved by the decompositions of these temporal states and the result is the following set of primitive patterns represented algebraically as {(1,3)e1s1, (1,3)e1w1, (1,3)e1p1, (1,3)p1a1, (4,4)e1s1, (4,4)*C*(e1w1), (4,4)e1p1, (4,4)p1a1, (5,5)e1s1, (5,5)*C*(e1w1), (5,5)e1p1, (5,5)p1a2, (6,6)e1s2, (6,6)e1w1, (6,6)e1p1, (6,6)p1a2}.

6.2.4 Snapshot of Temporal Association Pattern Instance

A TAPI with a time interval (or domain) T can be viewed as a collection of TAPIs at every t, where t is a time point in T. Each TAPI at t is called a snapshot of the TAPI. For example, the snapshot of the TAPI of Figure 6-4 at time 1 is represented algebraically as {(1,1)e1s1, (1,1)e1w1, (1,1)e1p1, (1,1)p1a1}.

6.2.5 Temporal Association Pattern Set

A Temporal Association Pattern Set (TAPS) α is a set of TAPIs. Therefore, a TAPS α is a set of sets of primitive patterns, where each "set element" represents the decomposition of a TAPI in α into its algebraic representation. A TAPS α is said to be homogeneous iff the snapshots of all the TAPIs α_i of α at t (denoted as α_i^t), for all $t \epsilon T$, are of the same intensional pattern; that is, the object instances in these patterns are members of the same object classes. A TAPS is heterogeneous if, for any $t \epsilon T$, there exist i and j, where $i \neq j$, such that α_i^t and α_j^t do not have the same intensional pattern. Figure 6-8(a) shows an example of a TAPS which contains three TAPIs about persons p1, p2, p3 and their associations with temporal object instances of Employee, Work_On and Project classes. The TAPIs in Figure 6-8(a) are the union of the TAPI in Figure 6-2 and the two TAPIs in Figure 6-8(b). Tabular representations of the TAPIs in this TAPS is given in Figure 6-8(c) for easier understanding. Since the TAPI of p2 does not involve (or is not associated with) any object instance in Employee class, it is not considered to be the same intensional pattern as the TAPIs of p1 and p3; thus, the TAPS in Figure 6-8 is heterogeneous.

6.2.6 Snapshot of Temporal Association Pattern Set

A snapshot of a TAPS α at t denoted by α^t contains the snapshots of each TAPI α_i at t (denoted by α_i^t) in α. A snapshot of a TAPS α at t is also a TAPS except the time interval is limited to t; it contains all the TAPIs of instance associations that are true at time t.

6.3 Temporal Association Algebraic Operators

The following notations will be used for the discussion of the operators of TA-algebra.

A, B, ..., K	Denote object classes.
CL_i	Denotes a variable for an object class.
$[R(CL_1, CL_2)]$	Denotes the association between classes CL_1 and CL_2, which can be explicitly named by an attribute.
$\{R(CL_1, CL_2)\}$	Denotes the set of temporal Inter-patterns having the association denoted by $[R(CL_1, CL_2)]$.
$\{R^t(CL_1, CL_2)\}$	Denotes the snapshot at t of the set of temporal Inter-patterns having the association denoted by $[R(CL_1, CL_2)]$.
$(s,t)a_i$	Denotes the ith temporal Inner-pattern (or instance) of class A which is valid during s and t.
a_i	Denotes IID of the ith temporal object instance of class A.
a_i^t	Denotes the snapshot of a_i at t. a_i^t can also be expressed as $(t,t)a_i$.
@	Denotes a temporal Inner-pattern variable.
$(s,t)a_ib_j$	Denotes a temporal Inter-pattern between a_i of class A and b_j of class B which is valid during s and t. In other words, a_i is associated with b_j during s and t.
$(s,t)C(a_ib_j)$	Denotes a temporal Complement-pattern between a_i of class A and b_j of class B which is valid during s and t (a_i is not associated with b_j during the time period).
$(s,t)D(a_ic_k)$	Denotes a temporal Derived Inter-pattern from class A to class C which is valid during s and t (a_i and c_k are indirectly associated with each other during the time period).
$(s,t)DC(a_ic_k)$	Denotes a temporal Derived Complement-pattern from class A to class C which is valid during s and t (a_i and c_k are indirectly non-associated with each other during the time period).
α, β, r	Denotes TAPSs each contains a set of TAPIs.
α_i	Denotes the ith TAPI of the TAPS α.
α_i^t	Denotes the snapshot at t of the ith pattern of the TAPS α.
α^t	Denotes the snapshot at t of the TAPS α.
{W}, {X}, {Y},..	Denote sets of classes. Hence, $\alpha_{\{X\}}$ represents TAPS α which has temporal Inner-pattern(s) from the classes in {X}.

£(o)	Denotes the function which returns the valid time intervals of a temporal object instance o.
s(T)	Denotes the function which returns the start time of T.
e(T)	Denotes the function which returns the end time of T.

Temporal association algebra (TA-Algebra) operators are defined based on snapshot semantics. A TAPS α can be defined in terms of snapshot semantics in the following expression:

"$\alpha = U\alpha^t$ for $t \epsilon T$, where α^t is the snapshot of α at t and T is the time interval of α."

That is, the TAPS α is the union of the snapshot TAPSs α^t for all $t \epsilon T$. In the following, we describe the mathematical properties of TA-algebra operators. We shall illustrate the operations of these operators by examples using the TOG in Figure 6-8(a) as the database. All the TAPSs in the examples will be represented in terms of the five types of temporal primitive patterns.

6.3.1 T-Associate (*)

Temporal Associate (or T-Associate) operator is a binary operator (similar in function to relational join) which constructs a TAPS of complex patterns by concatenating the TAPIs of two operand TAPSs if they are related through a specified association according to what are stored in the OO database. That is, the snapshot at t, for any $t \epsilon T$, of each TAPI in the resulting TAPS is a concatenation of the snapshots at t of two TAPIs, one from each operand, if a specified association between these two patterns is satisfied. Since a TAPI may involve many classes and an object class may have more than one association with another class, it is necessary to specify through which association the concatenation of two TAPIs is intended. The result of T-Associate operation is a TAPS containing no duplicate. T-Associate operator can be viewed as a navigational operator and is used for applications when a user is interested in the relationships of object instances of the classes involved in the path, e.g., whether a

person is an employee, whether an employee participates in a particular project, etc. A T-Associate operation on two TAPSs α and β over the association R between classes A and B is defined as follows:

$r = \alpha * [R(A, B)]\ \beta = \bigcup(r^t)$ for all t in T, where $r^t = \alpha^t * [R(A, B)]\ \beta^t = \{r_k^t \mid r_k^t = \alpha_i^t \cup \beta_j^t \cup \{(t,t)a_mb_n\} \wedge (t,t)a_mb_n \in \{R^t(A,B)\} \wedge (t,t)a_m \in \alpha_i^t \wedge (t,t)b_n \in \beta_j^t\}$

In the above expression, r_k^t is constructed by linking two TAPIs α_i^t and β_j^t through the association $(t,t)a_mb_n$, where k ranges over the total number of TAPIs that can be constructed from α and β over the association between classes A and B, $(t,t)a_mb_n$ is an instance of the association R between classes A and B, $(t,t)a_m$ is a member of α_i^t, and $(t,t)b_n$ is a member of β_j^t. Since each TAPI can be represented as a set of primitive patterns containing no duplicate, r_k^t is the set of primitive patterns constructed from the union of α_i^t, β_j^t, and the association $\{(t,t)a_mb_n\}$.

The association name [R(A,B)] following the operator can be omitted if the following conditions hold: (1) both α and β are Temporal Algebraic Expressions (TAE), (2) T-Associate operator operates on the last class in the linear expression of α and the first class in the linear expression of β and (3) there is a unique association between classes A and B.

An example of T-Associate operation is given in Table 6-1. It operates on two operand TAPSs α and β which represent TAPIs of the period T[1,NOW] of Person and Employee classes, respectively. Both α and β are represented in algebraic expressions of the primitive temporal patterns. Each TAPI in the resulting TAPS of T-Associate operation represents a pattern instance that "a person who is an employee during some time". Those persons who are not employees at any time will be dropped from the result. For example, if Figure 6-8(a) represents the contents of the database, p1 is associated with e1 from time 1 to NOW and p3 is associated with e3 from 1 to NOW,

they both will appear in the resulting TAPS r as temporal Inter-Patterns {(1,3)p1e1, (4,4)p1e1, (5,5)p1e1, (6,7)p1e1, (8,NOW)p1e1} and {(1,8)p3e3, (9,NOW)p3e3}, respectively; p2 (i.e., α_2) will be dropped because it does not have an association with any object instance in the TAPS β. The resulting TAPS r thus can be represented algebraically as {{(1,3)p1e1, (4,4)p1e1, (5,5)p1e1, (6,7)p1e1, (8,NOW)p1e1}, {(1,8)p3e3, (9,NOW)p3e3}}.

T-Associate (*) operator is commutative and conditionally associative as defined below:

α * [R(A,B)] β = β * [R(A,B)] α (commutativity)

($\alpha_{\{X\}}$ * [R(A,B)] $\beta_{\{Y\}}$) * [R(C,D)] $r_{\{Z\}}$ (associativity)
= $\alpha_{\{X\}}$ * [R(A,B)] ($\beta_{\{Y\}}$ * [R(C,D)] $r_{\{Z\}}$) (if C$\notin${X} and B$\notin${Z})

As stated above, the associativity holds true if α and r do not have temporal Inner-pattern of classes C an B, respectively. Thus, α will have no effect on the operation *[R(C,D)] on the left-hand side and r will have no effect on the operation *[R(A,B)] on the right-hand side. A detailed proof of this property can be found in [SU93]. The following is an example illustrating that the associativity will not hold if α contains a temporal Inner-pattern of class C. In this example, we assume that α = { {(t,t)a1b1, (t,t)b1c2} }, β = { {(t,t)b1c1} }, r = { {(t,t)d1} }, and the domain of the temporal algebra is as shown in Figure 6-9. Then,

(α * [R(A,B)] β) * [R(C,D)] r
= ({ {(t,t)a1b1, (t,t)b1c2} } * [R(A,B)] { {(t,t)b1c1} }) *[R(C,D)] { {(t,t)d1} }
= { {(t,t)a1b1, (t,t)b1c2, (t,t)b1c1} } *[R(C,D)] { {(t,t)d1} }
= { {(t,t)a1b1, (t,t)b1c2, (t,t)b1c1, (t,t)c2d1} }

and

α * [R(A,B)] (β * [R(C,D)] r)
= { {(t,t)a1b1, (t,t)b1c2} } * [R(A,B)] ({ {(t,t)b1c1} } *[R(C,D)] { {(t,t)d1} })
= { {(t,t)a1b1, (t,t)b1c2} } * [R(A,B)] {$\emptyset$} (i.e., since (t,t)c1d1 doesn't exist)
= {$\emptyset$}

6.3.2 TA-Complement(|)

Temporal Association Complement (TA-Complement) operator is a binary operator which concatenates a pair of TAPIs of two operand TAPSs α and β over a temporal Complement-pattern if a specified association does not exist between the TAPIs. That is, the snapshot pattern instance α_i^t in α will be concatenated with the snapshot pattern instance β_j^t in β through a complement pattern if α_i^t is not associated with β_j^t through the specified association. The operator is used to identify those patterns in two TAPSs that are not connected through a specified association, e.g., a person is not associated with (or hired as) an employee, an employee does not work on a project, etc. If each operand α and β is defined by a single class, then the operator is used to identify the pairs of object instances from two classes which are not associated with each other during some time period. The TA-Complement operator with snapshot semantics is defined as follows:

$r = \alpha \mid [R(A, B)]\ \beta = \bigcup(r^t)$ for all t in T, where $r^t = \alpha^t \mid [R(A, B)]\ \beta^t =$

$\{r_k^t \mid r_k^t = \alpha_i^t \cup \beta_j^t \cup \{(t,t)C(a_m b_n)\} \wedge (t,t)C(a_m b_n) \in \{R^t(A,B)\} \wedge (t,t)a_m \in \alpha_i^t \wedge (t,t)b_n \in \beta_j^t$
or $r_k^t = \alpha_i^t$: Exist $(t,t)a_m \in \alpha_i^t \wedge$ Not Exist $(t,t)b_n \in \beta^t$
or $r_k^t = \beta_j^t$: Exist $(t,t)b_n \in \beta_j^t \wedge$ Not Exist $(t,t)a_m \in \alpha^t$)

The result of TA-Complement operator is a TAPS containing TAPIs (i.e., r_ks) formed by concatenating one TAPI from each operand through a complement instance $C(a_m b_n)$ between classes A and B. In the case if α^t (or β^t) is empty or does not contain an object instance of class A (or B) at t, all patterns of β^t (or α^t) that contain an object instance of B(or A) at t are retained in the resulting TAPS.

An example of TA-Complement operation is shown in Table 6-2. It operates on the non-association between Employee and Work_On classes of the period T[1,NOW] represented by the TAPSs α and β, respectively. The result is a TAPS containing those TAPIs that "an employee is not associated with a particular Work_On instance (i.e., an

employee does not work on a particular project)". According to the TOG in Figure 6-8(a), e1 is not associated with w1 during T[4,5], e1 is not associated with w2 during T[1,NOW], e3 is not associated with w1 during T[1, NOW] and e3 is not associated with w3 during T[6,NOW][2]. Complement links are thus constructed between the corresponding versions of e1 and w1, e1 and w2, e3 and w1, and e3 and w3. The result of the TA-complement operation between Employee and Work_On is the TAPS *r* containing four temporal complement patterns. TA-Complement (|) operator is commutative and associative. For the similar reason described for the T-Associate operator, the associativity holds true conditionally.

$\alpha \mid [R(A,B)]\ \beta = \beta \mid [R(A,B)]\ \alpha$ (commutativity)

$(\alpha_{\{X\}} \mid [R(A,B)]\ \beta_{\{Y\}}) \mid [R(C,D)]\ r_{\{Z\}}$ (associativity)

$= \alpha_{\{X\}} \mid [R(A,B)]\ (\beta_{\{Y\}} \mid [R(C,D)]\ r_{\{Z\}})$ (if $C \notin \{X\}$ and $B \notin \{Z\}$)

6.3.3 TA-Select (δ):

Temporal Association Select (or TA-Select) is a unary operator which operates on a TAPS α to produce a subset of TAPIs that satisfy a specified predicate expression *P* within a time domain *TD*. $P = T_1\Theta_1T_2\Theta_2 \ldots \Theta_{n-1}T_n$, where T_i can be a term specifying a data condition, expressing the connectivity between two object classes, specifying the existence of an object instance in a pattern. Θ_i is a boolean operator (^ or v). *TD* is a specification of an operating time domain, either a time interval or a time point. The time information specified in *TD* can be explicit such as "1983 to 1986" or implicit such as "before John became a Manager". An extensive discussion on the specification of *TD* will be given in Section 6.5. Both predicate *P* and time domain *TD* are optional in a TA-Select operation. If only *P* is given, TA-Select operation will operate on all the

[2]Note, non-association between e3 and w3 does not exist until time 6 because w3 was not created yet before time 6.

temporal data of the operand based on the selection condition *P* (that is, *TD* is defaulted to the time domain of the operand); if only *TD* is given, TA-Select returns all the temporal data of the operand which fall into the time domain specified by *TD*[3]; if both *P* and *TD* are given, TA-Select will operate, based on *P*, only on the temporal data in the operand which fall into the time domain specified by *TD*. TA-Select operator with snapshot semantics is defined as follows:

$r = \delta(\alpha)[(TD)P] = \bigcup(r^t)$ for all t in *TD*, where
$r^t = \delta(\alpha)[(t)P] = \delta(\alpha^t)[P] = \{r_k^t \mid r_k^t = \alpha_i^t$ if $P(\alpha_i^t)$ is True$\}$

In the above expression, the statement "$P(\alpha_i^t)$ is True" means that α_i^t satisfies the predicate expression *P*. An example of TA-Select operation is given in Table 6-3. It operates on a TAPS α (where α is defined as "Person*Employee*Salary" of T[1,NOW]) with predicate *P* being "Employee.Salary = s4" and *TD* as T[9,11] to select those TAPIs of the period T[9,11] which represent that a person is an employee whose salary is s4. Since the operating time domain in this example is limited by to T[9,11], temporal data out of this time domain will not be considered. According to the TOG in Figure 6-8(a), during T[9,11] only employee e3 satisfies the predicate (i.e., e3 is with s4 during T[9,11]); therefore, the TAPI α_2 of T[9,11] will be retained and the TAPI α_1 is dropped because it does not satisfy the predicate during T[9,11].

6.3.4 TA-Project (π)

TA-Project operator which is similar to the relational Project operator is used to project over some subpatterns of a set of TAPIs it operates on. The temporal subpatterns to be projected are specified in the expression $E=\{e_1, e_2, \ldots e_n\}$. An optional path specification $C=\{c_1, c_2, \ldots, c_n\}$, which is a set of ordered sets of classes, should also

---

[3]In this case, TA-Select operation is similar to a time-slice operation.

be included in a projection operation to explicitly specify the path(s) to be retained in case a TAPI has multiple paths between the projected subpatterns. If no path specification is given, all the paths between the projected subpatterns will be retained. If a path in the original TAPI consists of all temporal Inter-patterns, a temporal D-Inter-pattern is added to the projected subpatterns in the result to indicate that the projected subpatterns are associated with each other. Otherwise, a temporal D-complement-pattern is added to specify the projected subpatterns are not associated with each other. TA-Project operator with snapshot semantics is defined as follows:

$r = \pi(\alpha)[E,C] = \bigcup(r^t)$ for all t in T, where $r^t = \pi(\alpha^t)[E,C] =$
$\{r_k^t \mid r_k^t = \pi(\alpha_i^t)[\{e_i\},\{c_i\}]$ where $\{e_i\}=E$, $\{c_i\}=C$ and the projected subpatterns $\{e_i\}$ of α_i^t are linked by the path specified in $\{c_i\}$ $\}$

The above expression states that r_k^t consists of the projected subpatterns $\{e_i\}$ of α_i^t and $\{e_i\}$ are connected through the paths specified in $\{c_i\}$. Table 6-4 shows an example of TA-Project operation. It operates on a TAPS α (which is defined as "Person*Employee*Work_On*Project") over subpatterns Person*Employee, and Project. The result of TA-Project operation in this example is a TAPS which contains the TAPIs showing the direct relationships between persons and employees and indirect relationships between employees and projects (i.e., the relationships that persons as employees are indirectly associated with projects). No path specification is necessary in this example because there is only one path between Employee and Project. The resulting TAPS r consists of two TAPIs: the first TAPI represents the fact that p1 as e1 participated in pj1 during T[1,3] and participated in pj1 and pj3 during T[6,7] and T[8,NOW], respectively; the second TAPI represents the fact that p3 as e3 participated in pj2 during T[1,8] and T[9,NOW]. Links between instances of Employee and Project

classes in both TAPIs are Derived Inter-Patterns, which preserve the association information of these non-adjacent instances.

6.3.5 T-NonAssociate (!)

Temporal NonAssociate (or T-NonAssociate) operator is a binary operator used to identify the temporal association patterns α_is in α that are not associated with any TAPI β_j in β via the specified association, and those β_js in β that are not associated with any TAPI α_i in α via the same association. Each pattern in the resultant TAPS is formed by concatenating two patterns, one from each operand, via a complement pattern. T-NonAssociate operator is used for identifying, as an example, those employees who are not associated with any project and projects which are not associated with any employee. T-NonAssociate operation with snapshot semantics is defined as follows:

$r = \alpha\ !\ [R(A, B)]\ \beta = \bigcup(r^t)$ for all t in T, where $r^t = \alpha^t\ !\ [R(A, B)]\ \beta^t =$
$\{r_k^t \mid r_k^t = \alpha_i^t \cup \beta_j^t \cup \{(t,t)C(a_mb_n)\} \wedge (t,t)C(a_mb_n) \in \{R^t(A,B)\} \wedge (t,t)a_m \in \alpha_i^t \wedge (t,t)b_n \in \beta_j^t \wedge$
ForAll $((t,t)a_mb_j$ and $(t,t)a_ib_n$ in the algebra domain) $((t,t)a_m \notin \alpha^t \wedge (t,t)b_n \notin \beta^t)$
or $r_k^t = \alpha_i^t$ if Exist$((t,t)a_m \in \alpha_i^t) \wedge$ Not Exist$((t,t)b_n \in \beta^t)$
or if ForAll $((t,t)b_n \in \beta^t)$ Exist$(a_k, a_k \neq a_m)((t,t)a_k \in \alpha^t \wedge (t,t)a_kb_n \in R^t\{(A,B)\})$
or $r_k^t = \beta_j^t$ if Exist$((t,t)b_n \in \beta_j^t) \wedge$ Not Exist$((t,t)a_m \in \alpha^t)$
or if ForAll $((t,t)a_m \in \alpha^t)$ Exist$(b_k, b_k \neq b_n)((t,t)b_k \in \beta^t \wedge (t,t)a_mb_k \in R^t\{(A,B)\})$ $\}$

In the above expression, r_k^t is formed by concatenating α_i^t and β_j^t via the temporal Complement-pattern $(t,t)C(a_mb_n)$ under the condition that α_i^t is not associated with any β_j^t and vice versa. The IID variables a_i and b_j used in the above expression are a member of α but α^t and a member of β but β^t, respectively. In the special case when the snapshot patterns of α (or β) at $t \in T$ have Inner-patterns of A (or B) and can not be concatenated with any snapshot pattern of β (or α) at t, these snapshot patterns of α (or β) at t will be retained in the result if one of the following three conditions hold: (1) β^t (or α^t) is an empty TAPS, (2) all TAPIs of β^t (or α^t) do not have temporal Inner-patterns

of B (or A), or (3) all TAPIs of β^t (or α^t) which have temporal Inner-patterns of B (or A) have associations with TAPIs of α^t (or β^t).

Table 6-5 shows an example of T-NonAssociate operation. It operates on two operand TAPSs α and β, where α is defined as "Person[IID$\neq$p1]" and β is defined as "Employee". The result of this operation is a set of TAPIs representing those persons in α who are not employees in β and those employees in β who are not the persons specified in α. In this example, since p3 (i.e., α_2) is associated with e3 (i.e., β_2) as shown in Figure 6-8, they both will be dropped. p2 and e1, however, are not associated with any pattern in β and α, respectively. They will be retained. The resulting TAPS r thus contains only one temporal Complement-Pattern constructed between p2 and e1.

T-NonAssociate (!) operator is commutative but not associative:

α ! [R(A,B)] β = β ! [R(A,B)] α (commutativity)

6.3.6 TA-Intersect (•)

Temporal Association Intersect (or TA-Intersect) operator is used to construct a TAPI from two TAPIs which share a common subpattern. TA-Intersect operator is conceptually equivalent to the JOIN operator in the relational algebra except that the TA-Intersect combines two operands if they have a common subpattern instead of a common attribute value. It operates on two operand TAPSs over a set of specified classes {W}. Each TAPI in the resulting TAPS is constructed by concatenating two TAPIs, one from each operand, which have a common subpattern specified by classes {W} and are valid at the same time. Those TAPIs with a common subpattern but different valid time will not be retained in the result. TA-Intersect operator with the snapshot semantics is defined as follows:

$r = \alpha_{\{X\}} \bullet \{W\}\ \beta_{\{Y\}} = \bigcup(r^t)$ for all t in T, where $r^t = \alpha^t_{\{X\}} \bullet \{W\}\ \beta^t_{\{Y\}} =$
$\{r_k^t \mid r_k^t = \alpha_i^t \cup \beta_j^t$ if (ForAll classes @$\in$ {W} in α_i^t, @$\in \beta_j^t$) ^
(ForAll classes @$\in$ {W} in β_j^t, @$\in \alpha_i^t$) }

The above expression states that r_k^t is formed (i.e., unioned) from α_i^t and β_j^t if they have common pattern in the classes specified in {W}. An example of TA-Intersect operation is given in Table 6-6. In this example, α is defined as "Person * Employee * Work_On * Project[P# = pj1] during the time period T[6,NOW]" and β is defined as "Person * Employee * Work_On * Project[P# = pj2 or P# = pj3] during T[9,NOW]". As shown in the TOG of Figure 6-8(a), e1 works on pj1 during T[6,NOW], e1 works on pj3 during T[9,NOW] and e2 works on pj2 during T[9, NOW], therefore, the TAPS α contains one TAPI and β contains two TAPIs. After performing TA-Intersect between α and β over Person and Employee classes, the first TAPIs of α and β will be concatenated in the resulting TAPS r because of the match in p1 and e1; the second pattern of β, however, will be dropped because it does not have a counterpart in α. When α_1 and β_1 are concatenated as r_1 in r, the time domain associated with r_1 is time-sliced to T[9,NOW] according to snapshot semantics. The resulting TAPI r_1 represents the fact that p1 is associated with e1 who works on pj_1 and pj3 during T[9,NOW].

TA-Intersect ($\bullet$) operator is commutative, conditionally associative and idempotent:

$$\alpha \bullet_{\{W\}} \beta = \beta \bullet_{\{W\}} \alpha \qquad \text{(commutativity)}$$

$$(\alpha_{\{X\}} \bullet_{\{W1\}} \beta_{\{Y\}}) \bullet_{\{W2\}} r_{\{Z\}} = \alpha_{\{X\}} \bullet_{\{W1\}} (\beta_{\{Y\}} \bullet_{\{W2\}} r_{\{Z\}}) \qquad \text{(associativity)}$$

(if $(\{W_1\} - \{W_2\}) \cap \{Z\} = \emptyset \wedge (\{W_2\} - \{W_1\}) \cap \{X\} = \emptyset$)

$$\alpha \bullet \alpha = \alpha \text{ (if } \alpha \text{ is a homogeneous TAPS)} \qquad \text{(idempotency)}$$

If $(\{W_1\}-\{W_2\})\cap\{Z\} \neq \emptyset$ or $(\{W_2\}-\{W_1\})\cap\{X\} \neq \emptyset$, the associativity is not always true because there are cases in which a pattern of β which fails to intersect with any pattern of r, may succeed by first intersecting with a pattern of α in the operation $\bullet_{\{W1\}}$ and then intersecting with a pattern of r in the operation $\bullet_{\{W2\}}$. It is obvious that the idempotency for this operator only holds when α is a homogeneous TAPS.

6.3.7 NT-Intersect (○)

Non-Temporal Intersect (or NT-Intersect) is a binary operator used to retain those TAPIs in both operands which have common subpattern as specified in {W} regardless of their valid times. The resultant TAPS will be the union of the retained patterns from two TAPSs and can be further operated on by other operators such as TA-Select operation to retain only the patterns from one TAPS. NT-Intersect operator is defined as follows:

$\alpha \circ\{W\} \beta = \{ r_k \mid$ $r_k = \alpha_i$ or $r_k = \beta_j$ if Exist i,j,t_1, and t_2 such that α_i^{t1} and β_j^{t2} have common patterns in the classes specified in {W}}

NT-Intersect is a useful operation when an across-time reference of temporal data is necessary. For example, we may want to retrieve the information of those current employees who participate in project pj2 or pj3 and who also participated in pj1 during T[1,3]. This query can be expressed in two temporal contexts, Employee * Work_On * Project [P# = pj2 or P# = pj3] at NOW and Employee * Work_On * Project[P# = pj1] during T[1,3], connected by the NT-Intersect operator over the Employee class. The first expression returns a TAPS containing those current employees who work on pj2 or pj3 and the second expression returns a TAPS containing those employees of T[1,3] who worked on pj1. When NT-Intersect operation is performed between these two TAPSs, TAPIs of the operands which have a match in Employee class will be retained. The result is a set of TAPIs of some current employees who works on pj2 or pj3 and also worked on pj1 during T[1,3], and some employees of T[1,3] who worked on pj1 and have been working on pj2 or pj3 during T[8,NOW]. Since we are interested only in information of those current employees in this query, we can further apply the TA-Select operator on the resulting TAPS r as $\delta(r)$[(NOW)]. Table 6-7 shows the detail of this example: α contains two TAPIs because e1 works on pj3 during

T[8,NOW] and e2 works on pj2 during T[1,NOW] and β contains one TAPI because only e1 worked on pj1 during T[1,3]. Since α_1 and β_1 has a match on e1, they both will be retained after the NT-Intersect operation; α_2 is dropped because it does not have a match in β.

NT-Intersect ($\bigcirc$) operator is commutative, associative and conditionally idempotent:

$\alpha \bigcirc_{\{W\}} \beta = \beta \bigcirc_{\{W\}} \alpha$ (commutativity)
$(\alpha_{\{X\}} \bigcirc_{\{W1\}} \beta_{\{Y\}}) \bigcirc_{\{W2\}} r_{\{Z\}} = \alpha_{\{X\}} \bigcirc_{\{W1\}} (\beta_{\{Y\}} \bigcirc_{\{W2\}} r_{\{Z\}})$ (associativity)
$\alpha \bigcirc \alpha = \alpha$ (if α is a homogeneous TAPS) (idempotency)

NT-Intersect is not idempotent if the TAPS α is not homogeneous. For example, if α = {{(1,1)a1b1}, {(1,1)c1d1}}, then the NT-Intersect operation $\alpha \bigcirc_{\{A\}} \alpha$ will result in a TAPS α'={{(1,1)a1b1}} and the NT-Intersect operation $\alpha \bigcirc_{\{C\}} \alpha$ will result in a TAPS α''={{(1,1)c1d1}}, where α' and α'' are not equal to α.

6.3.8 TA-UNION (+)

Temporal Association Union (or TA-Union) is a binary operator which combines two TAPSs into one. Union-compatibility is not a requirement in TA-algebra. The operands can be either heterogeneous or homogeneous TAPSs. In this operation, if the pattern of a TAPI α_i in one operand is identical to that of a TAPI β_j in the other operand, α_i and β_j will be combined into a single pattern in the result and their time intervals will be coalesced. TA-Union operator with snapshot semantics is defined as follows:

$r = \alpha + \beta = \bigcup(r^t)$ for all t in T, where $r^t = \alpha^t + \beta^t = \{r_k^t \mid r_k^t \epsilon \alpha^t \text{ or } r_k^t \epsilon \beta^t\}$

Table 6-8 shows an example of TA-Union operation. It operates on the two TAPSs α and β of Table 6-7. Since no TAPIs are identical, the resulting TAPS contains three TAPIs. TA-Union (+) operator is commutative, associative and idempotent:

$\alpha + \beta = \beta + \alpha$ (commutativity)
$(\alpha + \beta) + r = \alpha + (\beta + r)$ (associativity)
$\alpha + \alpha = \alpha$ (idempotency)

6.3.9 TA-Difference (-)

Temporal Association Difference (or TA-Difference) is a binary operator and the operands can be either heterogeneous or homogeneous TAPS. This operator implements the same concept as the DIFFERENCE operator in the relational algebra. It is used to eliminate the snapshots of TAPIs in the first TAPS if these snapshots of TAPIs also appear in the second TAPS. TA-Difference operator is defined as follows:

$r = \alpha - \beta = \bigcup(r^t)$ for all t in T, where
$r^t = \alpha^t - \beta^t = \{r_k^t \mid r_k^t = \alpha_i^t$ iff NotExist(β_j^t) such that $(\beta_j^t = \alpha_i^t)\}$

Table 6-9 shows an example of TA-Difference operation. In this example, α is the TAPS of all employees and β is the TAPS of the employees of T[1,8] who worked on a project. Performing the TA-Difference operation between α and β yields a TAPS which contains either TAPIs of those employees of T[9,NOW] or TAPIs of those employees who did not work on a project during T[1,8]. The resulting TAPS r contains two TAPIs: the first TAPI holds the information about e1 of T[9,NOW] which did not work on a project during T[4,5]; the second TAPI holds the information about e3 of T[9,NOW].

6.3.10 TA-Divide (÷)

TA-Divide (÷) operator implements the concept that a group of TAPIs with certain common features contains another set of TAPIs at some time points (or during some time intervals). For example, TA-Divide operator can be used in the case when

we are interested in those persons who, as employees, have salary s1 and live at address a1, etc. TA-Divide operation with snapshot semantics is defined as follows:

$r = \alpha \div_{\{W\}} \beta = \bigcup(r^t)$ for all t in T, where
$r^t = \alpha^t \div_{\{W\}} \beta^t = \{r_k^t \mid r_k^t = \alpha_i^{ts}$: ForAll(j) such that ($\beta_j^t$ is included in α_i^{ts}) }

α^{ts} in the above expression is a subset of the snapshot of α at t, which have common Inner-patterns for all classes of {W} and they together contain all patterns of the snapshot of β at t. If {W} is not specified, TA-Divide operation retains all the patterns of the snapshot of α at t for all $t \epsilon T$, each of which contains at least one pattern of the snapshot of β at t and they together contain all the patterns of the snapshot of β at t.

An example of TA-Divide is shown in Table 6-10. Suppose α is the union of the three TAPSs α', α", and α"' which are defined as "Person * Employee * Work_On * Project[P# = pj1]", "Person *Employee*Work_On*Project[P# = pj2]" and "Person * Employee * Work_On * Project[P# = pj3]", respectively, and β is the TAPS {{(15,30) pj1}, {(15,30)pj3}}. If we perform a TA-Divide operation between α and β with {W} defined to be {Person, Employee}, the result of this operation is a set of TAPIs each of which represents the fact that "a person as an employee worked on pj1 and pj3 simultaneously during T[15,30]". The retained TAPIs in r do not contain information before time 15 and after time 30 because β contains only the information of T[15,30] and is empty before 15 and after 30.

6.4 Specification of Time Domain

In TA-algebra, we provide three ways for specifying a time domain (or time intervals) of an operand in the TA-Select operation. They correspond to the time specifications introduced in Chapter 5: i.e., (1) explicit time specification, (2) data (or temporal relationship) specification, and (3) version sequence specification. In an

explicit time specification, a specific time is given in the form of a time point such as "seven o'clock last night" or a time interval such as "last three months". It is the easiest way of specifying a time domain and is used when an explicit time information is available. A data specification is used to specify a time domain based on the time some data hold true such as "when John worked on Project P1", "when Mary became a secretary", etc. A complex data condition or a temporal data relationship can also be used as the data specification for defining a time domain. For example, if we are interested in temporal data of the period when employee John's salary was greater than $60K "before Tom became a Manager", we can specify the time domain by using the interval comparison operator "BEFORE" as follows:

```
WHEN I := INTERVAL(Employee[Name=John, Salary > $60K])
        WHERE I BEFORE INTERVAL(Employee[Name=TOM, Title=Manager])
```

where "I := INTERVAL(Employee[Name = John, Salary > $60K])" in the above specification assigns the time interval(s) when John's salary was greater than $60K to the interval variable I. In TA-algebra, the sixteen temporal relationships (each is represented by an interval comparison operator) between two time intervals presented in Chapter 5 can also be decomposed into algebraic representations through the TA-Select operator for selecting historical object instances and the time functions £(), s() and e() for projecting the time domains of the selected historical instances. Here £() is a time function used to project the valid time interval of a historical version, s() is a time function used to project the start time of a time interval and e() is a time function used to project the end time of a time interval. For example, the specification of a time domain (TD) using BEFORE as stated above can be represented algebraically as follows:

α_1 = δ(Employee)[Name="John" ^ Salary=$60K]
α_2 = δ(Employee)[Name="Tom" ^ Title="Manager"]
TD = £(δ(α_1)[(e(£(α_1)) < s(£(α_2))]

The algebraic representation of a time domain using the other relationships can be derived similarly. Table 6-11 restates the semantics of the sixteen interval comparison operators using these functions.

Version sequence is another way used in TA-algebra to specify a time domain. For example, a user may be interested in a particular version of an object instance, such as the 4th change of Mary's Title, whose valid time information is not directly available. In this case, version sequence specification is used in TA-algebra to define the time domains for a temporal query through interval and version functions such as FIRST(), NTH(), etc.

6.5 Query Example

We have described ten TA-Algebra operators and their mathematical properties. In this section, we give some examples to demonstrate how these operators can be used to formulate temporal queries for processing an O-O temporal database. Alternative expressions for the same query can be formulated; however, it is the task of a query toptimizer to choose the best one for execution. In the following formulation of algebraic expressions, we will first give a specification of a temporal query in the high-level query language OQL/T and then the corresponding algebraic expression which is generated by a query translator. We shall decompose the specification of the algebraic expression for each query in the following examples into several steps for ease of understanding.

Example 6-1: What was John's salary when Mary was a clerk?

OQL/T representation:

```
WHEN INTERVAL(Employee[Name="Mary" ^ Title="Clerk"]
CONTEXT emp:Employee[Name="John"]
RETRIEVE emp.Salary
```

Algebraic representation:

α_1 = δ(Employee) [(Name = "Mary") ^ (Title = "Clerk")]
α_2 = δ(Employee) [(£(α_1)) Name="John"]
α_3 = π(α_2)[Salary]

In the first expression, we use Employee class as the operand for the TA-Select operator which has a selection predicate containing two terms "Name = Mary" and "Title = Clerk" connected by a boolean operator " ^ ". A time domain specification is not given in the first expression because we are interested in all the instances when Mary became a Clerk. The selected TAPIs that Mary was a Clerk are assigned to α_1 which is used to define the time domain in the second expression. The second expression selects the TAPIs with "Name = John" during the period when Mary was a Clerk (i.e., *TD* in TA-Select is specified by £(α_1)). The result is assigned to α_2. In the third expression, we perform TA-Project operation on α_2 to project the Salary attribute. The result is assigned to α_3 which contains John's salary when Mary was a Clerk.

Example 6-2: What were the salaries of those employees who worked on Projects P5 and P6 when Mary was a Clerk during the period of T[1, 6]?

OQL/T representation:

```
WHEN I:INTERVAL(Employee[Name="Mary" ^ Title="Clerk"])
        WHERE I WITHIN T[1, 6]
CONTEXT  emp:Employee AND (*Work_On * Project[P#=p5],
                           *Work_On*Project[P#=p6])
RETRIEVE emp.Salary
```

Algebraic representation:

α_1 = δ(Employee)[Name="Mary" ^ Title="Clerk"]
TD = £(δ(α_1)[(s(£(α_1)) >= 1 ^ e(£(α_1)) <= 6)]
α_2 = δ(Employee)[(TD)] * δ(Work_On)[(TD)] * δ(Project)[(TD) P#=P5]
α_3 = δ(Employee)[(TD)] * δ(Work_On)[(TD)] * δ(Project)[(TD) P#=P6]
α_4 = α_2 $\bullet_{\{Employee\}}$ α_3
α_5 = π(α_4)[Salary]

In the first expression, we select the TAPI(s) with Name equal to Mary and Title equal to Clerk from Employee class and assign the result to α_1. Since we are interested in the historical information of employee's salaries during the periods when the temporal

relationship "WITHIN" holds between the valid time intervals associated with α_1 and the time interval T[1, 6], we retrieve those qualified time intervals through the second expression and assign the result to TD. In the third and fourth expressions, we perform TA-Select operations on the TAPSs defined by the TAPIs "Employee * Work_On * Project[P# = P5]" and "Employee * Work_On * Project[P# = P6]" over the time domain TD and assign the results to α_2 and α_3, respectively. We then perform a TA-Intersect operation between α_2 and α_3 over Employee class in the fifth expression and assign the result to α_4. Up to this point, we have already constructed the pattern that the query is interested in: "employees who worked on Projects P5 and P6 when Mary was a Clerk during T[1, 6]". The last step is to derive answer by performing a TA-Project operation on α_4 over the Salary attribute. The result is assigned to α_5.

Example 6-3: Retrieve the names of those current employees who ever worked on Project p1 during the period T[8, 12] and whose salary were greater than $30K during T[3, 6].

OQL/T representation:

```
CONTEXT emp:Employee
        INTERSECT (Employee)
WHEN T[8, 12]
CONTEXT Employee * Work_On * Project[P# = p1]
        INTERSECT (Employee)
WHEN T[3, 7]
CONTEXT  Employee[Salary > $30K]
RETRIEVE emp.Name
```

Algebraic representation:

TD_1 = T[8, 12]
TD_2 = T[3, 7]
$\alpha_1 = \delta(\text{Employee})[(\text{NOW})]$
$\alpha_2 = \delta(\text{Employee})[(TD_1)] * \delta(\text{Work_On})[(TD_1)] * \delta(\text{Project})[(TD_1)\ P\# = p1]$
$\alpha_3 = \delta(\text{Employee})[(TD_2)\ \text{Salary} > \$30K]$
$\alpha_4 = \alpha_1 \bigcirc_{\{\text{Employee}\}} \alpha_2 \bigcirc_{\{\text{Employee}\}} \alpha_3$
$\alpha_5 = \pi(\delta(\alpha_4)[(\text{NOW})])[\text{Name}]$

This query retrieves the names of those current employees who satisfy some historical conditions. The first expression selects those current employees from

Employee class and the result is assigned to α_1. The second expression selects the TAPIs that "an employee work on p1" of the period T[8,12] and the result is assigned to α_2. The third expression selects those employees of the period T[3,7] whose salaries are greater than \$30K and the result is assigned to α_3. We then perform NT-Intersect operations among α_1, α_2 and α_3 over Employee class to retain those TAPIs of α_1, α_2 and α_3 that have the common Inner-pattern of Employee class. The result is assigned to α_4. Since we are only interested in the information of current employees, we can simply perform a TA-Select operation on α_4 with time domain limited to NOW and use TA-Project to project over the Name attribute in the last expression. The result is assigned to α_5.

6.6 Comparison with Related Work

Recent efforts on the design of temporal algebra have been focused on extending the relational algebra [CLI85, GAD86, CLI87, GAD88, TAN86, LOR88, TUZ90, SAR90] because of the simplicity and well developed theories of the relational model. There are three major approaches taken in these efforts: (1) retaining the semantics of all the standard operators [LOR88] (2) changing the semantics of all the standard operators [CLI85, CLI87, GAD88, TUZ90], and (3) changing the semantics of part of the standard operators [TAN86, NAV89, SAR90]. In the first approach, time is treated simply as another attribute of a tuple during data processing. The advantage of this approach is that it is simple extension. However, meaningless intermediate results (such as "a tuple existed at t" which nevertheless is not defined in the domain of an application) may be generated in this approach because the semantics of time is not considered by the system when temporal data are processed. As a result, extra operators such as fold, unfold, extend, etc. need to be provided to process the time attribute in

order to filter out those irrelevant intermediate results. The main disadvantages in this approach, therefore, are the cumbersome composition of algebraic expressions for a temporal query and the inefficiency in query processing. In the second approach, time is incorporated into the standard operators based on either lifespan or snapshot semantics. Meaningless intermediate results will not be generated in this approach. Therefore, only the operators such as tdom, time-slice etc. which are used to define the temporal domain for an operand need to be provided for processing temporal data in this approach. Composition of algebraic expressions for temporal queries is simpler than the first approach while the implementation is much more complex. In the third approach, time factor is considered only for some of the standard operators such as "select" but not for the others such as "join". Illegal data types will be generated in this approach because a tuple resulted from join operation(s) will contain more time notions than what is defined originally. As a result, the closure property is not retained. A summary and evaluation of the recent efforts on relational temporal algebra are reported in [McK91].

In addition to the efforts in extending relational algebra, Rose and Segev [ROS92] have proposed a temporal object-oriented algebra (TO-Algebra) for an object-oriented database. In TO-Algebra, each operator is associated with a time interval which defines the temporal domain of the operand(s) to be operated on. The processing of temporal data in TO-Algebra is based on the concepts of Time Sequence and lifespan of homogeneous and heterogeneous object graphs which are defined as a set of object instances with the same and with different attributes, respectively. Each Time Sequence itself in TO-algebra is modeled as an object which grows as its associated object instance evolves. The lifespan of an object instance can be derived from the Time Sequence object. However, the objects in TO-algebra are not modeled uniformly. For example, an ordinary object is associated with a Time Sequence for capturing its evolution; while a

Time Sequence, which is also treated as an object in the model and keeps growing (or evolving) as its associated object instance evolves, is not associated with a Time Sequence to describe or capture its own growth/history. That is, the Time Sequence object is not modeled in the same way as an ordinary object. Furthermore, the operators in TO-algebra is more expressive than the high-level operators in their proposed object-oriented temporal language [ROS91] which is an extension of SQL.

Rose & Segev also claimed that TO-algebra has the closure property in the sense that it operates on one or more collections of objects and values and returns a collection of objects and values. However, the collection of objects and values returned from the TO-algebra may be of a different type from the collection of objects and values in the operand(s) because TO-algebra supports multiple time notions. For example, if the notions of transaction time and valid time are supported in TO-algebra, each (legal) data unit will be associated with one transaction time and one valid time. Nevertheless, the resultant objects and values returned from a join operation (i.e., Temporal Intersection) in TO-algebra will contain more than one transaction time notions because the transaction time notion of one operand may be different from the other. As a result, the returned objects and values are of a different type from the operands each of which is associated with one transaction time. TO-algebra thus will not be able to retain the closure property from the view point of data type according to [McK91].

Unlike TO-algebra, the operators in TA-algebra do not have a time specification (except TA-Select). They operate on the time domain of each operand. If a particular time domain for an operand is desired, the TA-Select operation can be used to limit the temporal operating range. By doing so, we are able to make the operators of TA-algebra more primitive and less expressive than the high-level query language OQL/T proposed in [SU91]. TA-algebra also supports the processing of heterogeneous as well

as homogeneous object patterns. The heterogenous patterns supported in TA-algebra, however, are more general than the heterogeneous objects supported in TO-algebra [ROS92] since the heterogeneous patterns of the former can contain different types of attributes as well as different types of associations (or relationships).

TA-algebra is at least functionally equivalent to both the temporal algebras proposed for the relational model and the TO-algebra proposed for an object-oriented model because it supports the operations supported by these algebras. For example, the semantics of the "rollback" operator proposed in [McK90, ROS92] can be captured in TA-algebra by the TA-Select operation with the deactivation of proper temporal knowledge rules; the "select-when" and the "select-if" operators proposed in [CLI87, ROS92] can be captured in TA-algebra simply by the TA-Select operation.

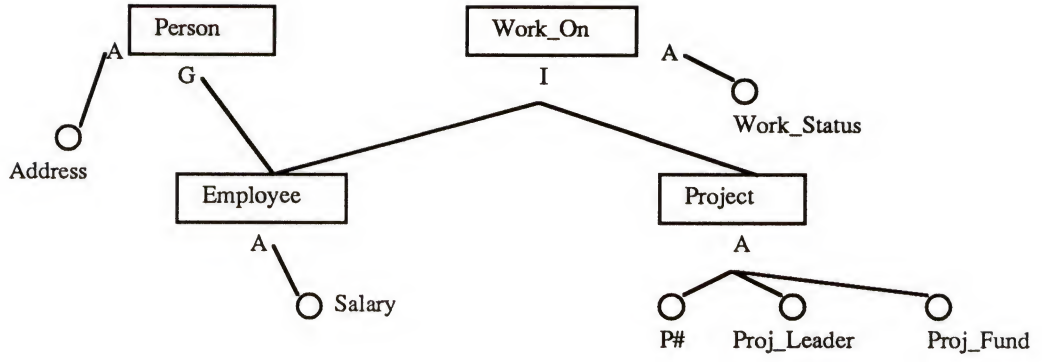


Figure 6-1: Schema graph of a company database.

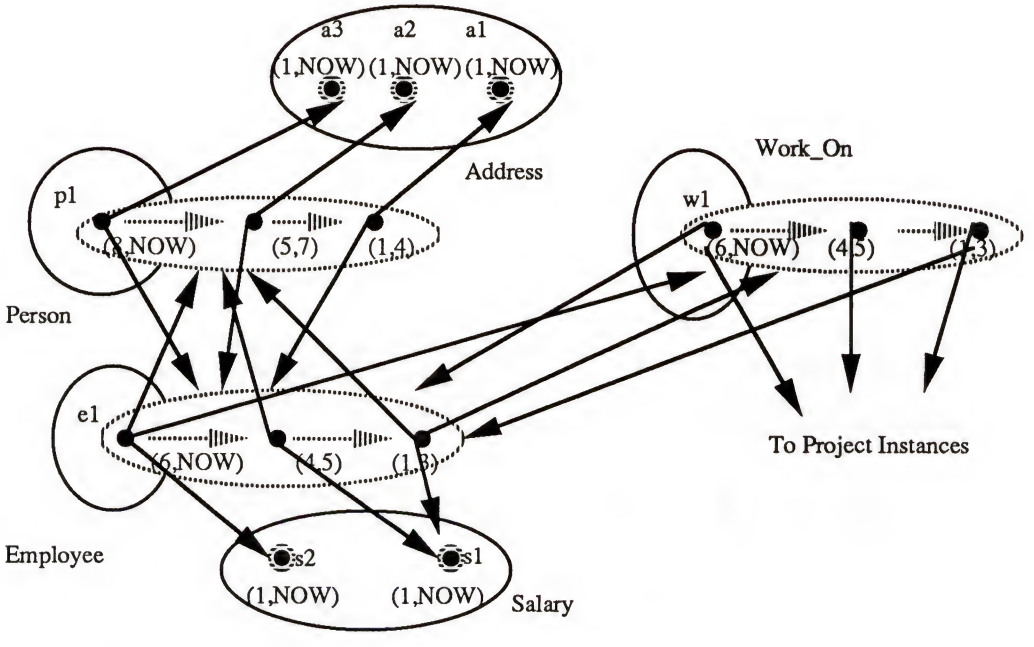


Figure 6-2: Temporal Object Graph for a partial schema of the company database.

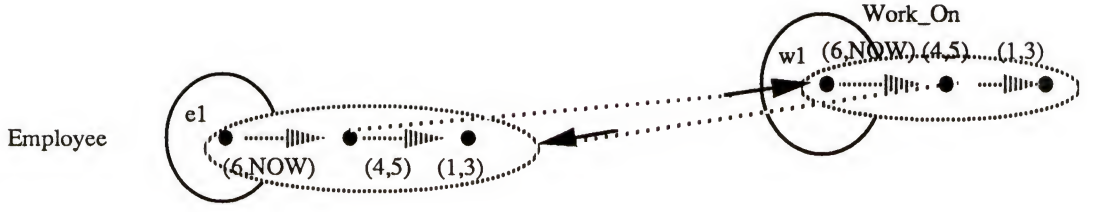


Figure 6-3: Illustration of the non-association between e1 and w1 during T[4,5].

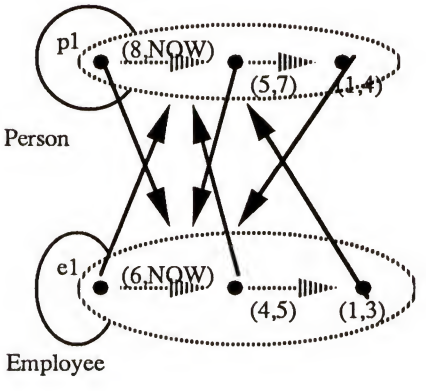


Figure 6-4: Temporal pattern of p1*e1 extracted from Figure 6-2.

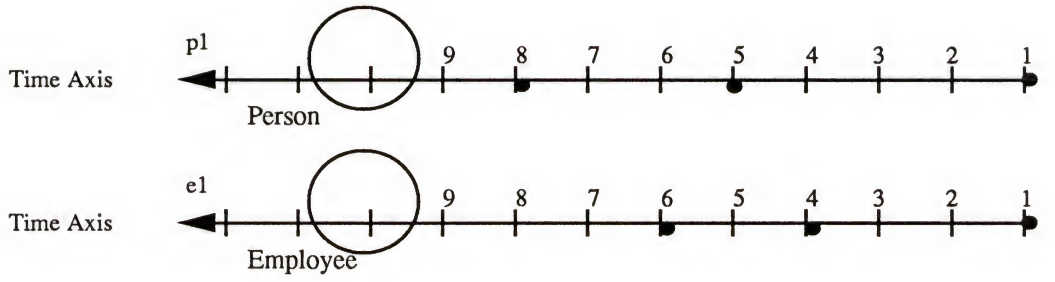


Figure 6-5: Redrawing of Figure 6-4 with Time Axis.

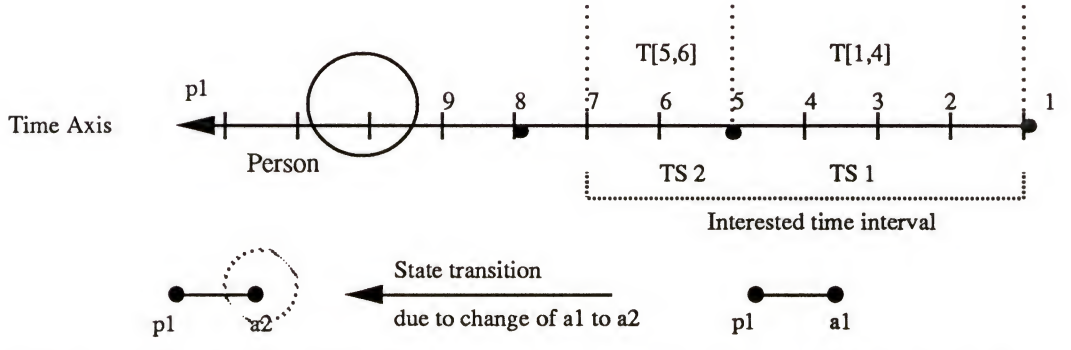
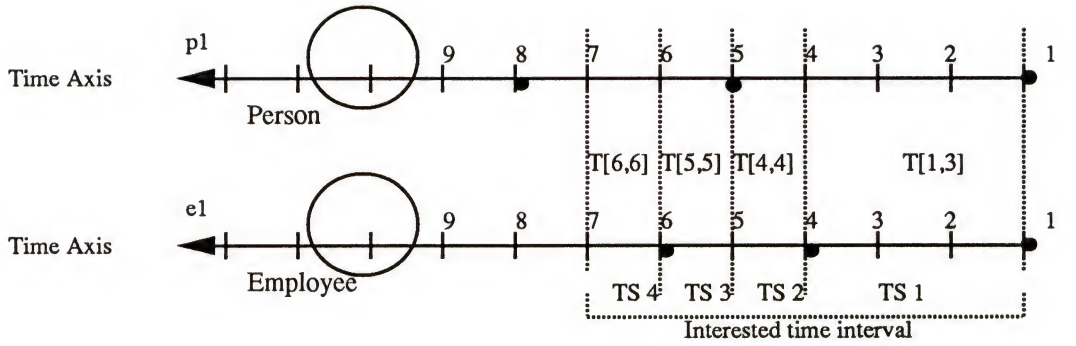
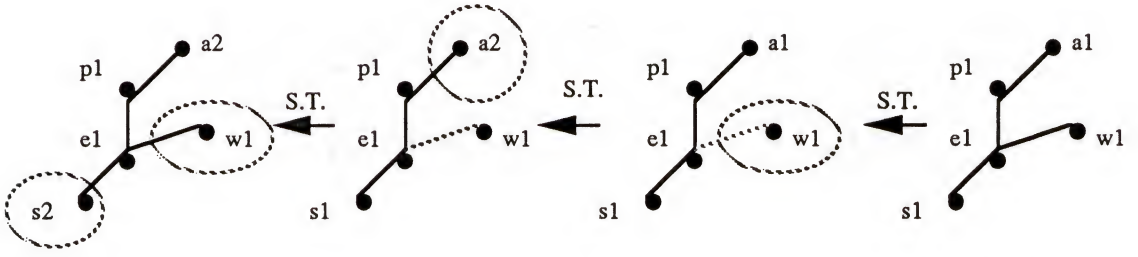


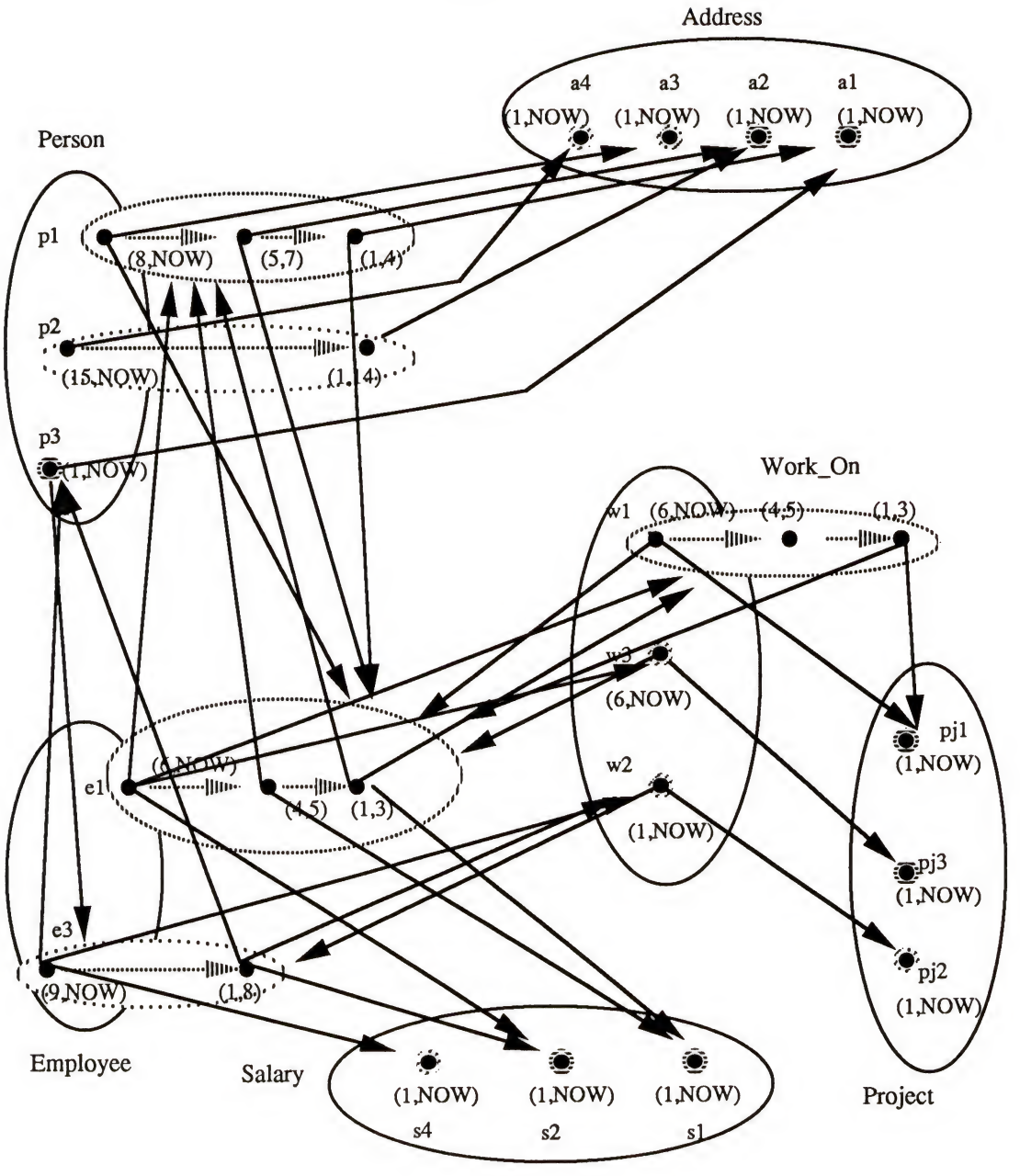
Figure 6-6: Temporal states of the TAPI p1 during T[1,6].





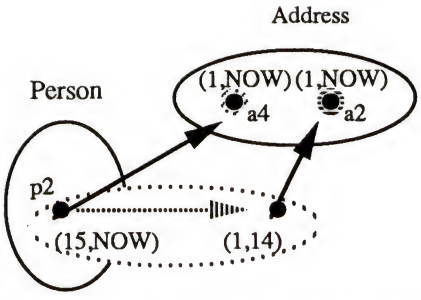
T[6,6]: TS 4 of p1e1. T[5,5]: TS 3 of p1e1. T[4,4]: TS 2 of p1e1. T[1,3]: TS 1 of p1e1.

Figure 6-7: Temporal states of the TAPI p1*e1 during T[1,6].

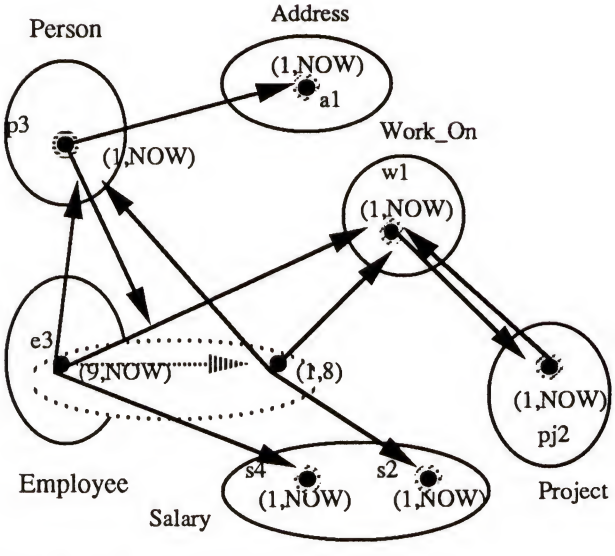


(a)

Figure 6-8: Temporal Object Graph for partial schema of the company database.
a) The TOG consisting of the temporal patterns of Figure 6-2 and Figure 6-8(b);
b) The temporal patterns of p2 and p3; c) Tabular representation of the TOG.



The temporal pattern p2 and its associated object instances and attribute values.



The temporal pattern p3 and its associated object instances and attribute values.

(b)

Figure 6-8: continued.

Person class

Start	End	IID	Address	associated objects
1	4	p1	a1	e1
5	7	p1	a2	e1
8	NOW	p1	a3	e1
1	14	p2	a2	Null
15	NOW	p2	a4	Null
1	NOW	p3	a1	e3

Employee class

Start	End	IID	Salary	associated objects
1	3	e1	s1	p1, w1
4	5	e1	s1	p1
6	NOW	e1	s2	p1, w1, w3
1	8	e3	s2	p3, w2
9	NOW	e3	s4	p3, w2

Work_On class

Start	End	IID	associated objects
1	3	w1	pj1, e1
6	NOW	w1	pj1, e1
1	NOW	w2	pj2, e3
6	NOW	w3	pj3, e1

Project class

Start	End	IID	associated objects
1	NOW	pj1	w1
1	NOW	pj2	w2
6	NOW	pj3	w3

(c)

Figure 6-8: continued.

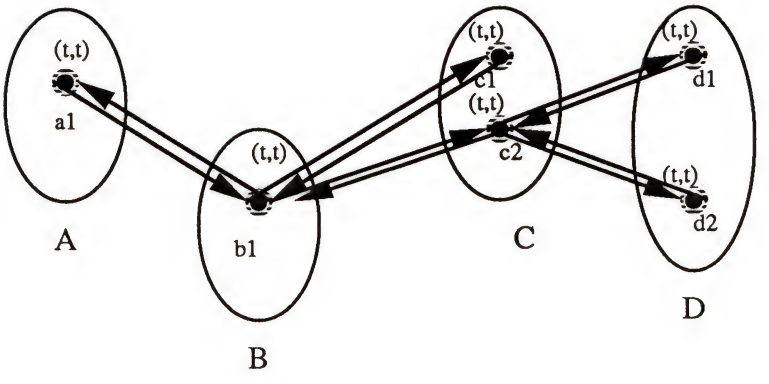


Figure 6-9 An example of temporal algebra domain

Table 6-1: T-Associate operation (*).

Let α: Person of T[1,NOW] and β: Employee of T[1,NOW]
That is
α = { {(1,4)p1, (5,7)p1, (8,NOW)p1}, {(1,14)p2, (15, NOW)p2}, {(1,NOW)p3} }
β = { {(1,3)e1, (4,5)e1, (6,NOW)e1}, {(1,8)e3, (9,NOW)e3} }
If $r = \alpha * \beta$
Then r = { {(1,3)p1e1, (4,4)p1e1, (5,5)p1e1, (6,7)p1e1, (8,NOW)p1e1},
{(1,8)p3e3, (9,NOW)p3e3} }

Table 6-2: TA-Complement operation (|).

Let α: Employee of T[1,NOW] and β: Work_On of T[1,NOW]
That is
α = { {(1,3)e1, (4,5)e1, (6,NOW)e1}, {(1,8)e3, (9,NOW)e3} }
β = { {(1,3)w1, (6,NOW)w1}, {(1,NOW)w2}, {(6,NOW)w3} }
If $r = \alpha \mid \beta$
Then r ={ {(4,5)*C*(e1w1)}, {(1,3)*C*(e1w2), (4,5)*C*(e1w2) (6,NOW)*C*(e1w2)},
{(1,3)*C*(e3w1), (4,5)*C*(e3w1),(6,8)*C*(e3w1), (9,NOW)*C*(e3w1)},
{(6,8)*C*(e3w3), (9,NOW)*C*(e3w3)} }

Table 6-3: TA-Select operation (δ).

Let α: Person*Employee*Salary of T[1,NOW]
That is
α = { {(1,3)p1e1, (1,3)e1s1, (4,5)p1e1, (4,5)e1s1, (6,NOW)p1e1, (6,NOW)e1s2},
{(1,8)p3e3, (1,8)e3s2, (9,NOW)p3e3, (9,NOW)e3s4} }
If $r = \delta(\alpha)$ [(T[9,11])Employee[Salary = s4]]
Then r = { {(9,11)p3e3, (9,11)e3s4} }

Table 6-4: TA-Project (π).

Let α: Person*Employee*Work_On*Project of T[1,NOW]
That is
α = { {(1,3)p1e1, (1,3)e1w1, (1,3)w1pj1, (6,7)p1e1, (6,7)e1w1, (6,7)e1w3, (6,7)w1pj1, (6,7)w3pj3, (8,NOW)p1e1, (8,NOW)e1w1, (8,NOW)e1w3, (8,NOW)w1pj1, (8,NOW)w3pj3}, {(1,8)p3e3, (1,8)e3w2, (1,8)w2pj2, (9,NOW)p3e3, (9,NOW)e3w2, (9,NOW)w2pj2} }
If $r = \pi(\alpha)$ [Person*Employee, Project;]
Then r = { { (1,3)p1e1, (1,3)D(e1pj1), (6,7)p1e1, (6,7)D(e1pj1), (6,7)D(e1pj3), (8,NOW)p1e1, (8,NOW)D(e1pj1), (8,NOW)D(e1pj3)}, {(1,8)p3e3, (1,8)D(e3pj2), (9,NOW)p3e3, (9,NOW)D(e3pj2)} }

Table 6-5: T-NonAssociate (!).

Let α: Person[IID$\neq$p1] of T[1,NOW]
β: Employee of T[1,NOW]
That is
α = { {(1,14)p2, (15,NOW)p2}, {(1,NOW)p3} }
β = { {(1,3)e1, (4,5)e1, (6,NOW)e1}, {(1,8)e3, (9,NOW)e3} }
If $r = \alpha \,!\, \beta$
Then r = { {(1,3)C(p2e1), (4,5)C(p2e1), (6,14)C(p2e1) (15,NOW)C(p2e1)} }

Table 6-6: TA-Intersect ($\bullet$).

Let α: Person*Employee*Work_On*Project[P# = pj1] of T[6,NOW]
β: Person*Employee*Work_On*Project[P# = pj2 or pj3] of T[9,NOW]
That is
α = { {(6,7)p1e1, (6,7)e1w1, (6,7)w1pj1, (8,NOW)p1e1, (8,NOW)e1w1, (8,NOW)w1pj1} }
β = { {(9,NOW)p1e1, (9,NOW)e1w3, (9,NOW)w3pj3}, {(9,NOW)p3e3, (9,NOW)e3w2, (9,NOW)w2pj2} }
If $r = \alpha \bullet_{\{\text{Person, Employee}\}} \beta$
Then r = { {(9,NOW)p1e1, (9,NOW)e1w3, (9,NOW)w3pj3, (9,NOW)e1w1, (9,NOW)w1pj1} }

Table 6-7: NT-Intersect (○).

Let α: Employee*Work_On*Project[P# = pj2 or pj3] at NOW
β: Employee*Work_On*Project[P# = pj1] of T[1,3]
That is
α = { {(NOW,NOW)e1w3, (NOW,NOW)w3pj3},
{(NOW,NOW)e3w2, (NOW,NOW)w2pj2} }
β = { {(1,3)e1w1, (1,3)w1pj1} }
If $r = \alpha \bigcirc_{\{Employee\}} \beta$
Then r = { {(NOW,NOW)e1w3, (NOW,NOW)w3pj3}, {(1,3)e1w1, (1,3)w1pj1} }

Table 6-8: TA-UNION (+).

Let α: Employee*Work_On*Project[P# = pj2 or pj3] at NOW
β: Employee*Work_On*Project[P# = pj1] of T[1,3]
That is
α = { {(NOW,NOW)e1w3, (NOW,NOW)w3pj3},
{(NOW,NOW)e3w2, (NOW,NOW)w2pj2} }
β = { {(1,3)e1w1, (1,3)w1pj1} }
If $r = \alpha + \beta$
Then r = { {(NOW,NOW)e1w3, (NOW,NOW)w3pj3},
{(NOW,NOW)e3w2, (NOW,NOW)w2pj2}, {(1,3)e1w1, (1,3)w1pj1} }

Table 6-9: TA-Difference (-).

Let α: Employee of T[1,NOW]
β: $\pi(\delta$(Employee*Work_On*Project) [(T[1,8])]) [Employee]
That is
α = { {(1,3)e1, (4,5)e1, (6,NOW)e1}, {(1,8)e3, (9,NOW)e3} }
β = { {(1,3)e1, (6,8)e1}, {(1,8)e3} }
If $r = \alpha - \beta$
Then r = { {(4,5)e1, (9,NOW)e1}, {(9,NOW)e3} }

Table 6-10: TA-Divide (÷).

Let α': Person*Employee*Work_On*Project[P# = pj1] of T[1,NOW]
α'': Person*Employee*Work_On*Project[P# = pj2] of T[1,NOW]
α''': Person*Employee*Work_On*Project[P# = pj3] of T[1,NOW]
$\alpha = \alpha' + \alpha'' + \alpha'''$
β = { {(15,30)pj1}, {(15,30)pj3} }

That is

α' = { {(1,3)p1e1, (1,3)e1w1, (1,3)w1pj1, (6,7)p1e1, (6,7)e1w1, (6,7)w1pj1, (8,NOW)p1e1, (8,NOW)e1w1, (8,NOW)w1pj1} }
α'' ={ {(1,8)p3e3, (1,8)e3w2, (1,8)w2pj2, (9,NOW)p3e3, (9,NOW)e3w2, (9,NOW)w2pj2}}
α'''={ {(6,7)p1e1, (6,7)e1w3, (6,7)w3pj3, (8,NOW)p1e1, (8,NOW)e1w3, (8,NOW)w3pj3}}

and α = { {(1,3)p1e1, (1,3)e1w1, (1,3)w1pj1, (6,7)p1e1, (6,7)e1w1, (6,7)w1pj1, (8,NOW)p1e1, (8,NOW)e1w1, (8,NOW)w1pj1},
{(1,8)p3e3, (1,8)e3w2, (1,8)w2pj2, (9,NOW)p3e3, (9,NOW)e3w2, (9,NOW)w2pj2},
{(6,7)p1e1, (6,7)e1w3, (6,7)w3pj3, (8,NOW)p1e1, (8,NOW)e1w3, (8,NOW)w3pj3} }

If $r = \alpha \div_{\{person,\ Employee\}} \beta$
Then r = { {(15,30)p1e1, (15,30)e1w1, (15,30)w1pj1}, {(15,30)p1e1, (15,30)e1w3, (15,30)w3pj3} }

Table 6-11: The interval comparison operators and their representations in TA-Algebra operators.

Relationship between two time intervals specified by Interval Comparison Operator:	Conditions when the temporal relationship is True:
£(A) BEFORE £(B) or £(B) AFTER £(A)	End time of £(A) is less than Start time of £(B) (i.e., e(£(A)) < s(£(B))
£(A) PRECEDE £(B) or £(B) FOLLOW £(A)	End time of £(A) is equal to Start time of £(B) (i.e., e(£(A)) = s(£(B))
£(A) P-CROSS £(B) or £(B) F-CROSS £(A)	End time of £(A) is greater than Start time of £(B) and Start time of £(A) is less than Start time of £(B) (i.e., e(£(A)) > s(£(B)) ^ s(£(A)) < s(£(B))
£(A) EQUAL £(B)	Start time of £(A) is equal to Start time of £(B) and End time of £(A) is equal to End time of £(B) (i.e., s(£(A)) = s(£(B)) ^ e(£(A)) = e(£(B)))
£(A) L-CONTAIN £(B) or £(B) L-WITHIN £(A)	Start time of £(A) is equal to Start time of £(B) and End time of £(A) is greater than End time of £(B) (i.e., s(£(A)) = s(£(B)) ^ e(£(A)) > e(£(B)))
£(A) O-CONTAIN £(B) or £(B) I-WITHIN £(A)	Start time of £(A) is less than Start time of £(B) and End time of £(A) is greater than End time of £(B) (i.e., s(£(A)) < s(£(B)) ^ e(£(A)) > e(£(B)))
£(A) R-CONTAIN £(B) or £(B) R-WITHIN £(A)	Start-time of £(A) is less than Start-time of £(B) and End-time of £(A) is equal to End-time of £(B) (i.e., s(£(A)) < s(£(B)) ^ e(£(A)) = e(£(B)))
£(A) CROSS £(B)	£(A) (P-CROSS or F-CROSS) £(B)
£(A) CONTAIN £(B)	£(A) (L-CONTAIN or O-CONTAIN or R-CONTAIN) £(B)
£(A) WITHIN £(B)	£(A) (L-WITHIN or O-WITHIN or R-WITHIN) £(B)

CHAPTER 7
A DELTA-INSTANCE MULTI-SNAPSHOT STORAGE MODEL

This chapter describes our effort in the design and evaluation of physical temporal databases. We present a Delta-Instance Multi-Snapshot (DIMS) storage model which takes into consideration the trade-off between storage consumption and processing efficiency. The DIMS storage model achieves storage saving by integrating the advantages of attribute time-stamping and synchronous attributes. In this model, temporal data are compressed by using the technique of "delta file" to reduce data redundancy. Temporal data in DIMS are partitioned into blocks based on their time intervals to expedite their access and processing. Moreover, a snapshot of the most recent temporal data of all the object instances in each partitioned data block is stored to avoid lengthy traversals during the materialization of temporal data. Since the amount of temporal data in each block is fixed, the retrieval cost of temporal data in the proposed model is negligibly affected by the growth of temporal database. Access of temporal data in DIMS is further speeded up by a multi-layered index mechanism using conventional index trees. In this work, we compare our proposed DIMS model and its associated multi-layered index mechanism with the existing storage models and indexing techniques in order to understand their relative merits. The comparison is based on the following three aspects: (1) storage consumption of temporal data, (2) time for temporal data materialization, and (3) processing times for a set of benchmark queries.

This chapter is organized as follows. In Section 1, we present the DIMS storage model. Rationale for each design consideration in the DIMS is described. In Section 2,

we describe the storage models of the attribute and the tuple time-stamping approaches and the indexing techniques proposed in [AHN88, GUN90, ELM90] which will be compared with the multi-layered index mechanism of our proposed DIMS storage model. In Section 3, we describe a set of benchmark queries. In Section 4, we present the results of our analysis and comparison with respect to storage consumption, temporal data materialization and temporal data retrieval.

7.1 Delta-Instance Multi-Snapshot Storage Model

In this section, we present a Delta-Instance Multi-Snapshot (DIMS) storage model which is designed for efficient management of OO temporal knowledge bases. We first describe a general data structure for storing variable length temporal object instances. The use of the familiar "delta file" technique to store historical instances is then described. We then present the strategies for materializing temporal object instances using both forward and backward processes. To achieve efficient retrievals of temporal data, we propose a data-partitioning strategy which takes into considerations the pattern of temporal queries and the two-phase query processing strategy of OQL/T. In this data-partitioning strategy, time interval (or physical time window) is the primary partitioning criterion of temporal data. Within each partition of temporal data, we further use object versions and/or attribute values as the criteria to cluster those temporal data which are likely processed together. In order to avoid a lengthy traversal along a version chain during the process of temporal data materialization, we propose a multi-snapshot store which uses both fixed- and variable-storage formats for each cluster of temporal data. We also propose an access method with layered indices for efficient search of temporal data.

7.1.1 Data Format of Temporal Object Instances

In DIMS storage model, a temporal object instance is stored in the format shown in Figure 7-1. It consists of Start-time, End-time, IID, attribute-value pairs (Ai, Vi), and version pointers (both previous-version pointer (pvp) and next-version pointer (nvp) for bidirectional traversals of temporal object instances). The data format may vary in storage size to suit the sizes of temporal object instances of different classes.

7.1.2 Using Delta Instance Concept for the Management of Temporal Data

In order to save storage space, we adopt the "delta file" technique for managing temporal object instances by storing only the changed attributes. That is, an object instance is stored in the current database when it is initially created. As the object instance evolves, the changed attributes are recorded in a new version of object instance in the current database; while the old data of only the modified attributes together with the time information (called "*delta instance*") are stored as the old version of the object instance in the historical area[1]. The current version and historical versions of an object instance are linked together through the pvp and nvp pointers. An example of delta instance is given below. Assume that the object instance John of Employee class in this example is modeled by the instance identifier 01; it was created at time 1 and has been modified at time 4 and time 8, respectively on different attributes as illustrated below:

Start	End	IID	Dept	Salary	Title	
8	-	01	R&D	$60K	Engr	current version
4	7	01	R&D	$45K	Jr. Engr	second historical version
1	3	01	R&D	$30K	Jr. Engr	first historical version

[1] The delta instances of temporal data in the historical area are physically separated from current data to achieve search efficiency for the current data which are more frequently referenced.

Based on the proposed concept of delta instance, the evolution of the temporal data of employee John will be recorded and represented as in Figure 7-2(a), (b), and (c).

7.1.3 Materialization of Temporal Object Instance

One major concern in the design of a storage model using a compression technique (such as delta instance) is the materialization of historical object instances. In DIMS, each delta instance contains only a partial information of a historical object instance; some of the attribute values of the historical object instance are contained in and can be derived from its succeeding versions. In order to obtain the complete set of attribute values of a historical object instance, it is necessary to traverse through these successive historical versions. There are two possible ways to materialize historical object instances in DIMS: backward materialization and forward materialization.

7.1.3.1 Backward materialization

The backward materialization process, which can also be thought of as a replacement process, needs the support of pvp and traverses history from the newest version of an object instance. The newest version of an instance is fetched and stored in the memory. If only the current data is of interest, the traversal will not be necessary. Otherwise, a traversal through the history chain will take place. If any attribute value is found during the traversal back to the history through pvp, that attribute value will be used to replace the value of the corresponding attribute kept in the memory. This process continues until the desired temporal object instance has been reached. The attribute values in the memory are then the attribute values of the desired temporal object instance.

For example, if we are interested in the information of employee John at time 5, the system will first find the newest version of John which has complete attribute values

(IID 01), (Emp-Name, John), (Salary, $60K), and (Title, Engr). Because the newest object version started at time 8 which is later than what is interested, the system will follow the pvp to search for the previous version object instance. When the system finds the previous version historical object instance which contains two attribute values (Salary, $45K) and (Title, Jr. Engr.), these two attribute values will be used to replace (Salary, $60K) and (Title, Engr). The attribute values during time interval T[4,7] are (IID 01), (Emp-Name, John), (Salary, $45K), and (Title, Jr. Engr.) Since the time interval of this version contains the interested time point 5, the searching process stops and the attribute values of the temporal object instance John at time 5 will be (IID 01), (Emp-Name, John), (Salary, $45K), and (Title, Jr. Engr.)

7.1.3.2 Forward materialization

The forward materialization process, which can also be thought of as a filling-in process, needs the support of nvp and some time index for locating the interested temporal object instance directly. Since the accessed delta instance may contain only part of attribute values needed, the system will look forward (i.e., toward the direction of the current instance) and traverse its more recent versions using nvps to fill in the absent attribute values. Whenever an attribute of a more recent version, which is absent from the interested historical object instance, is encountered during the search, its value will be used to fill in the historical object instance and be kept in the memory. Those attributes which have already appeared in the temporal object instance will be ignored during the search.

We use the same example of the backward searching to illustrate the idea of forward searching. Through a time index, the system will quickly find the temporal object instance of John of the interval T[4,7] which contains attribute values (Salary, $45K) and (Title, Jr. Engr.) Because there are absent attributes in this temporal object

instance, the system needs to look forward by following the nvp pointers to search for its succeeding versions. When the first succeeding version object instance is found (which started at time 8 and actually is the newest version in this example), the attribute values (IID 01), (Emp-Name, John) will be used to fill in the interested temporal object instance and be kept in the memory, and the attributes (Salary, $60K) and (Title, Engr) will be ignored. At this point, all the absent attributes in the temporal object instance have been found; therefore, the searching process stops and the temporal object instance of John at time 5 will have the attribute values (IID 01), (Emp-Name, John), (Salary, $45K), and (Title, Jr. Engr.).

7.1.4 Partitioning of Temporal Data

Since it is inevitable that a temporal database will grow to a very large size due to the evolution of object instances, the performance of processing temporal data degrades accordingly. Although indices over temporal data using some conventional as well as new access methods [AHN86, GUN90, ELM91] have been proposed to alleviate the performance problem, processing of temporal data may still be inefficient for two reasons. First, the indices established for a temporal database are based on the entire sets of temporal instances in the database; therefore when the size of the temporal database grows, so do the indices. As a result, the cost of searching the indices increases logarithmically with respect to the increase of the temporal database size and the maintenance of these large indices can be very costly. Second, when these indices can not be used in a query processing, a costly sequential search through the whole temporal database would be required even though the desired temporal data may occupy only a small segment of the time line.

In order to avoid the above problems and achieve more efficient processing of temporal data, we propose a data partitioning strategy. It takes into considerations the pattern of temporal queries and the two-phase query processing strategy of OQL/T. Since time information is provided in OQL/T queries, we partition temporal data of object instances into blocks based on some selected time intervals (which are selected based on either *a priori* or *a posteriori* information) in a class-dependent manner[2]. That is, all the temporal data of a class within a time interval which are likely to be processed together are clustered into one Temporal Data Block (TDB) with its own local index. Each TDB[3] is a logical entity of temporal data and can be mapped to a number of fixed-size physical blocks in the secondary storage area. The database of the temporal KBMS generally can be partitioned into several TDBs and the number of TDBs increases monotonically as the knowledge base expands. Those long-lived object instances which persist across two consecutive TDBs will be duplicated in both TDBs with the time intervals being properly divided to fit the two blocks.

We use employee John's history as an example to illustrate the idea of partitioning temporal data. We firstly assume that time line is partitioned into intervals of T[1,5], T[6,14], T[15,32], etc., and then place all the historical data into its corresponding TDB. That is, the first TDB contains temporal data of the period T[1, 5], the second TDB contains temporal data of the period T[6, 14], and so on. The partitioning of temporal data of employee John is illustrated in Figure 7-3. In this example, object instance John's history is divided into two TDBs based on the given time

---

[2]The set of time intervals selected for partitioning temporal data into blocks is different from one class to another depending on the evolution rate of the object instances in each class.

[3]Since the amount of object instances in one class is different from that in another, the size of a logical TDB is also class-dependent.

intervals. Since the temporal object instance <4, 7, 01, John, $45K, Jr. Engr.> lives across TDB 1 (i.e. T[1, 5]) and TDB 2 (i.e. T[6, 14]), it will be replicated in both TDBs with the time interval being properly divided into T[6,7] and T[4,5][4].

By partitioning the instances of a class into blocks and associating each block of temporal data an (local) index, the search for temporal data can be focused on the relevant TDBs and their associated indices instead of the entire class and its index. It is definitely more efficient to search a smaller index established for the data of a TDB than an index over the whole class. In addition, the amount of temporal data and the size of the index are both fixed once a TDB is established; therefore, the search time associated with each TDB will be negligibly affected by the incoming new temporal data and thus can be considered as a constant. In the case when a query can not make use of the available indices, a sequential search through the TDB(s) that fits the time specification of the query is still much faster than sequentially searching through the entire set of object instances.

7.1.5 Clustering of Temporal Data

Data clustering is a technique of organizing data together based on some criteria so that those semantically related data which are potentially processed together can be physically adjacent. The purpose of this data organization technique is to reduce the disk access which is often the bottleneck of a data processing system. In DIMS, once a TDB is established, the amount and content of temporal data in the TDB do not change. Therefore, it is beneficial to organize the temporal data of a TDB in some format to

---

[4]Note, based on the Multi-Snapshot Store to be proposed later, the copy with T[4,5] will have complete attribute values because it is the latest version of the object instance John in the TDB 1, while the copy with T[6,7] only has the changed attribute values because it is the earliest version of object instance John in the TDB 2.

favor certain types of queries. Within each TDB, temporal data can be clustered based on the criteria of object version and/or attribute value. When object version is used as the clustering criterion, all the historical versions of an object instance within one TDB are physically adjacent and in chronological order. This choice of clustering temporal data is to favor those queries which ask for the versions of one object instance. When attribute value is used as the clustering criterion, those temporal object instances of one TDB whose attributes satisfy certain conditions will be gathered into physically adjacent area. This choice of clustering is to favor those queries which ask for temporal object instances satisfying some selection predicates. Since the materialization of temporal data stored as delta instances needs to traverse historical versions of an object instance in our proposed model, we choose to cluster temporal data based on object versions so that disk I/Os can be reduced during the materialization process.

7.1.6 Multi-Snapshot Store

Since temporal data are stored as "delta instances" and only current data contains a complete set of attribute values, materializations of object instances would involve long traversals of historical versions either in a backward or forward manner. In order to circumvent this problem during a temporal data materialization process, we propose the concept of Multi-Snapshot store which is used in conjunction with the partitioned store. In the Multi-Snapshot store, we store a snapshot (or a complete set of attribute values) of the most recent versions of all the object instances of a class in each TDB; the remaining earlier versions of those object instances are still stored as delta instances in the TDB. By doing so, we are making the information of temporal data in each TDB self-contained; therefore, it is not necessary to traverse from the most current version to materialize the temporal data of the historical versions. Instead, the traversal can simply

start directly from the snapshot of the TDB to previous versions through history chains. Besides, since the temporal data in each TDB is self-contained, the combined scheme of partitioned temporal data with the multi-snapshot store facilitates parallel processing of TDBs.

7.1.7 Implementation of TDBs

In the Multi-Snapshot store, we use a recent data storage area (RDSA) for storing the snapshot of the latest versions of object instances of each TDB and a historical data storage area (HDSA) for storing the delta instances of their earlier versions of the same TDB. Since the latest versions of object instances always contain complete sets of attribute values and the storage size of each attribute is fixed, RDSA has a fixed-size storage format. HDSA, on the other hand, uses a variable-size storage format because each delta instance contains only values of the changed attributes and the number of changed attributes in each delta instance can be different. Although the sizes of RDSA and HDSA for a TDB of a class can be different from those of another class, the sizes of all RDSAs and those of HDSAs for TDBs of one class are identical. In addition, RDSA and HDSA of one TDB do not have to be the same size and each can be implemented by a number of fixed-size physical buckets.

In forming TDBs for one class, we firstly allocate one RDSA and one HDSA for the first TDB. When data are initially created, they are placed in the RDSA. As data evolve, the original values of the changed attributes with the time information are shifted into the HDSA; the new attribute values with the new time information remain in the RDSA. When the specified time interval of the first TDB expires, those object instances in the RDSA which "live through" the TDB will be copied into a new (or the second) RDSA and their time intervals will be properly adjusted. This new RDSA and another

new HDSA will make up the second TDB. This process of forming TDBs continues as more data are entered into the knowledge base. Figure 7-4 illustrates the concept of forming TDBs for a class using the example of employee John's history. It also shows how temporal object instances which persist through TDBs are divided.

7.1.8 Layered Index of Temporal Data

Since temporal data in the proposed DIMS storage model are clustered and partitioned into TDBs, we also propose to keep an index tree of time intervals on these TDBs in the main memory (called TDB Index). TDB Index will provide quick access to any TDB. Since the index tree of time intervals is an append-only tree (i.e, deletion of an entry will not happen and time intervals in the index arrive in an ascending chronological order), it is reasonable to implement TDB Index with an AP-tree (adopted from [GUN90] and to be described in section 7.3) for reducing the index maintenance cost (e.g., entry-insertion and node-balancing) and for utilizing storage space more efficiently. The algorithm of TDB Index using an AP-tree is given in Appendix C.

Although TDBs can be quickly located through TDB Index, the amount of temporal data in each TDB may still be large; a sequential search through the TDBs for temporal data can still be very time-consuming. In this case, it may be desirable to build indices within each TDB by using a conventional access method (such as B-tree, B^+-tree, etc.) There are two possible approaches for building indices: one is to build indices for only the temporal object instances of the latest version of a TDB, and the other is to build indices for all the temporal object instances of a TDB. The first approach can index on IIDs and needs smaller index trees. The data search in this approach has to start from the latest version of each TDB using the backward materialization strategy. The other approach can index on selected attribute values and needs larger index trees.

This approach provides direct access to the desired historical versions of temporal object instances in each TDB; the complete set of attribute values of the desired historical version can be obtained by the forward materialization strategy.

7.2 Existing Storage Models and Access Methods

In this section, we describe the storage models for implementing the temporal databases using attribute and tuple time-stamping approaches. We also present some existing indexing techniques: Accession List proposed in [AHN88], AP-tree and its two variations proposed in [GUN90], and Time Index and two-level Time Index proposed in [ELM91]. We combined the existing storage models and the above proposed access methods and used them as the basis for evaluating DIMS.

7.2.1 Storage Models for Attribute and Tuple Time-Stamping Approaches

Attribute and tuple time-stampings are the two techniques commonly used to model the historical information of a temporal database. In tuple time-stamping, time information is augmented with each tuple while in attribute time-stamping, time information is augmented with each attribute. According to [LUM84, AHN86, SNO88, GUN90, JEN92b, JEN92d], the storage models for implementing the temporal databases using these two approaches are assumed to be the direct mappings from their logical representations. In addition, it is also assumed that a temporal database is append-only and the temporal data in an append-only database are stored in a chronological order (i.e., temporal data are sorted and stored by the time when they arrive the database). In this dissertation, we shall follow the conventional assumptions for the storage models of attribute and tuple time-stamping approaches in our performance comparison and evaluation.

Figures 7-5 and 7-6 illustrate the storage models for tuple and attribute time stamping approaches, respectively. In Figure 7-5, each historical version of a tuple is stored as it is logically represented. According to the way that temporal data are stored in an append-only database, the historical versions of one tuple may not be stored into physically adjacent spaces due to the fact that some other tuples may have evolved during the time interval between any two consecutive historical versions of the tuple. For instance, in Figure 7-5, the first and the second versions of the tuple with SSN equal to 0001 are physically separated by the first and the second versions of the tuple with SSN equal to 0002 and the first version of the tuple with SSN equal to 0003. The historical versions of one tuple, however, are linked through version pointers (i.e., nvp and/or pvp) so that all the versions of one tuple can be accessed efficiently. Since all the tuples in an append-only database are stored based on the time when they arrive the database, they are naturally sorted in a chronological order regardless of their surrogates.

Figure 7-6 shows that the evolutions of attributes in the attribute time-stamping approach are stored in a similar fashion as that of the evolutions of tuples in the tuple time-stamping approach, i.e., tuples of a TNF relation are stored and naturally sorted in chronological order regardless of their surrogates. However, the data of a N-attribute tuple are now scattered into N-1 TNF relations. Therefore, in attribute time-stamping, temporal joins [GUN90] are always required to materialize the temporal data of a tuple. Since temporal data of one tuple (or more precisely, one surrogate) in an append-only database are not necessarily stored in physically adjacent spaces, it is useful to pre-sort TNF relations based on surrogates before temporal joins are performed on them to gain processing efficiency (i.e., sort-merge join). Although it is possible that the TNF relations in the attribute time-stamping approach can be sorted based on surrogate and

time before they are stored in the secondary storage space to reduce materialization cost of temporal data, it would be very costly to maintain the storage model because each Insert or Update operation of a tuple will involve a reorganization of the entire relation. Reorganizing a relation is especially expensive when either the relation is large or the Update and the Insert frequencies are high. For this reason, we will not consider such a storage model. Detailed discussions of materializing temporal data in different storage models will be given in section 7.4.

7.2.2 Existing Indexing Techniques

In this section, we describe some existing indexing techniques. In order to illustrate the concept of these proposed indexing techniques, we use the data in Table 7-1 as an example. For expositional purpose, we also attach to each version of the object instances a version tag expressed as "e_{ij}", which stands for the jth version of the ith object instance. For example, e_{12} stands for the second version of the first object instance which is John's record during time 4 and 7, etc.

Accession List. Ahn and Snodgrass [AHN88] proposed Accession List as the access method for accessing temporal data in a temporal database. In their proposal, current data and historical data are physically separated and are stored in Current Store[5] and History Store, respectively. Data of an object instance in Current Store is chained through a pvp (previous version pointer) to the data of its historical versions in History Store; data of historical versions of the object instance in the History Store are also chained together through pvps in the manner that a later version points to its previous version. An index tree on IID or any attribute using a conventional index technique

[5]Some frequently accessed historical data may also be stored together with current data in Current Store.

(such as B^+-tree) is maintained for the data in Current Store. In addition, an Accession List which is a full index for all the historical versions of an object instance is maintained between the data in Current Store and those in History Store. Each entry in the Accession List of an object instance contains a pointer pointing to a historical version of the object instance. Search of historical data in the proposed approach always starts with the data in the Current Store. Once the data in the Current Store is located, historical data of previous versions can be accessed through the associated Accession List or pvp pointers. The retrieval cost of temporal data in this approach increases as the history grows and is proportional to the "age" of the retrieved data: the older the temporal data is to be retrieved, the more expensive is the access. Figure 7-7 shows the concept of Accession List for the example given in Table 7-1. In Figure 7-7, we assume e_{33} is a frequently accessed historical data and store it with other current data in the Current Store.

Time Index. Elmasri et al. [ELM90a] proposed Time Index (a B^+) tree as the access method for accessing temporal data of a temporal database. In this approach, an index point is created at the time point when (1) a new interval is started by an insert or update operation, or (2) an interval is terminated by an update or delete operation. Each index time point in this approach points to a bucket of pointers each of which points to a historical version of an object instance which is valid at the index time point. An illustration of a time index is given in Figure 7-8 in which the entry time point 8 is created because of the creations of e_{13} and e_{23} at time 8 and the entry time point 12 is created because of the deletion of e_{33} at time 11, etc.

Two-level Attribute/Time Index. Elmasri et al. [ELM90a] also proposed a two-level attribute/time index for those queries which involve predicates of both time and attribute values. The idea in this approach is to build a search tree consisting of a first

level index on attribute values (or object identifiers) and a second level Time Index for the evolution of each indexed attribute value (or object identifier). At the first level, an index entry is created whenever there is a new attribute value (or IID) created. At the second level, an index time point is created either at the time when an object instance gets the attribute value or at the time point after an object instance loses the attribute value. Each entry of the time index in the second level will point to a bucket of pointers each of which points to a historical version of the object instance which contains the indexed attribute value at the indexed time point. Examples of two-level Time-Index are given in Figures 7-9 and 7-10 in which Figure 7-9 uses object identifiers for the first-level index while Figure 7-10 uses salary values for the first level index.

ST-tree and AT-tree. Gunadhi and Segev [GUN90] proposed the AP-tree, the two-level ST-tree and the two-level AT-tree for supporting temporal queries which involve predicates of time, attribute values and object identifiers. AP-tree indexes on historical data based on the start time when a historical version of an object instance become valid. Therefore, in an AP-tree, an entry time point is created whenever a historical version of an object instance is created. Each entry in an AP-tree points to a set of historical versions whose valid time starts at the indexed time point. An AP-tree takes advantage of the "append-only" feature of a temporal database and is designed to fully utilize the storage space for an index tree[6]. However, since AP-tree is not efficient for processing temporal queries in most cases, the authors also introduced ST-tree and AT-tree to deal with queries which involve predicates of either "time and object

---

[6]According to [GUN90], the utilization of storage space for an AP-tree is close to 100%. However, the disadvantage accompanying the high storage-utilization of AP-tree is the increased complexity for deleting an entry point. Furthermore, AP-tree is not balanced: retrieving cost of one temporal data can be different from that of another temporal data. However, the retrieving cost of temporal data in AP-tree is less than or equal to that of a B^+-tree.

identifiers" or "time and attribute values". In the performance comparison of indexing techniques presented in the next section, we shall consider the ST-tree and AT-tree approaches. An ST-tree consists of a first level index (which is a B^+ tree) on all the object identifiers and a second level AP-tree for each object identifier on all the starting time points of the historical versions of an associated object instance. Each entry of the second level AP-tree points to the historical version of an object instance which started at the index time point. An AT-tree, which is similar to ST-tree, consists of a first level index on all the possible attribute values and a second level AP-tree with index on some particularly selected time points for minimizing page overflow. Each entry in the second level index points to a bucket of pointers each of which points to a historical version of an object instance which is instantiated with the index attribute value and has a valid time that overlaps with the time interval between the current and the succeeding index time points. Examples of AP-tree, ST-tree, and AT-tree are given in Figures 7-11, 7-12, and 7-13, respectively. In Figure 7-13, we have picked time points 5, 8, 10, and 15 as the entry points of a local (or second level) AP-tree for illustration purpose.

7.3 Benchmark Queries

Having presented our storage model and some existing techniques in the previous sections, we are interested in finding out their relative performances with respect to a set of benchmark temporal queries. In this section, we propose a set of benchmark queries which will be used for comparing different indexing techniques and storage models. A temporal query generally contains a specification of time (either time interval or time point) and some predicates of object identifiers and/or attribute values. Temporal data which fits the specified time will be retrieved and processed to determine if their identifiers or attribute values satisfy the predicate(s). Based on the above observation,

we divide temporal queries into two major categories: (time, IID-predicate) queries and (time attribute-predicate) queries. Queries of the first category are further divided into seven types and queries of the second category are further divided into three types as shown below:

Category I: (Time & IID-predicate) Queries,

Type1: retrieves the version of an object instance at a particular time point (i.e., (t1, IID)-type query).

Type2: retrieves the versions of an object instance within a time interval (i.e., ([t1, t2], IID)-type query).

Type3: retrieves the versions of all object instances at a given time point (i.e., (t1, IIDs)-type query)

Type4: retrieves the versions of all object instances within a time interval (i.e., ([t1, t2], IIDs)-type query)

Type5: retrieves the entire history of an object instance (i.e., ([1, NOW], IID)-type query).

Type6: retrieves the current version of an object instance (i.e., (NOW, IID)-type query)

Type7: retrieves the current version of all object instances (i.e., (NOW, IIDs)-type query).

Category II: (Time & Attribute-predicate) Queries,

Type8: retrieves the current data which satisfy some attribute-predicate (i.e., (NOW, attribute)-type query).

Type9: retrieves the historical data of a given time point which also satisfy some attribute-predicates (i.e., (t1, attribute)-type query).

Type10: retrieves the historical data of a given time interval which also satisfy some attribute-predicate (i.e., ([t1, t2], attribute)-type query).

7.4 Performance Analysis and Comparison

In this section, we present the results of a performance analysis and comparison between our proposed technique and the existing techniques in terms of storage consumption, cost of materialization of temporal data, and processing times for the benchmark queries. With respect to storage consumption, we compare the delta instance technique used in our approach (hereafter, we simply call it object instance time-stamping) against attribute and tuple time-stampings. With respect to the cost of materialization, we compare different techniques in terms of their materialization times for temporal data of both a specific time interval and the entire time line. Disk I/Os for retrieving the relevant temporal data as well as the CPU time for materializing the complete set of attribute values for each historical version of every object instance are considered. With respect to processing cost, we compare the performance of the layered indices used in the DIMS model against the access methods of Accession List, Time Index, two-level Time Index, and two-level ST and AT trees proposed in [AHN88, GUN90, ELM91].

In order to compare these different techniques, we develop analytical formulas for calculating the computational time and disk I/Os for all techniques. The developed formulas are given in Appendixes D to H. The notations and assumptions used in the analysis and comparison are as follows:

Notations:

N	The average number of attributes of each object class (or relation), among which, N-1 of them are non-synchronous and time-varying.
A	The average storage space in bytes for each attribute.
M	The average number of time-varying attributes involved in an evolution of each object instance. Each attribute has the same probability that it will be modified in each evolution of an object instance.
S	The total number of object instances (or surrogates) in an object class (or a relation).
N_{avg}	The average number of evolutions of an object instance (or surrogate) during an interval T.

T A sample time period.
TB The number of temporal data blocks during T, TB < N_{avg}.
C The number of object classes (relations) in a database schema.
Pg The size of a physical page.
n The number of TDBs involved in a specified time interval.
S_{avg} The average number of modified object instances at a time.

Assumptions:

Assumption 1: We assume that there are C classes (or relations) each of which has N attributes defined in the database schema. All N attributes are of the same size denoted by A and N-1 of them are time-variant. Time stamps have the same storage size A as ordinary attributes. Each object instance in a class will have N_{avg} evolutions on the average during a period T; each evolution of an object instance will involve the modifications of M attributes (where M < N). The probability P for each time-varying attribute of an object instance to be modified in each evolution is P = M/(N-1).

Assumption 2: For the DIMS model, we assume that the temporal data of the time interval T are evenly partitioned into TB number of temporal data blocks (TDBs) each of which may have a different time interval. That is, in each TDB, an object instance has N_{avg}/TB evolution on the average even though the time duration of one TDB may be different from that of another. In addition, we assume that the temporal data in each TDB are clustered and sorted based on IIDs; the historical versions of an object instance in each cluster are further sorted in a chronological order[7].

Assumption 3: Due to the computational overhead and the implementation difficulty for nested relations, we assume that the attribute time-stamping approach uses a storage model which stores temporal data in an append-only TNF relations as

[7]Clustering historical versions of object instances by IID's and sorting historical versions of an object instance in chronological order will guarantee an adequacy of a single disk scan during the materialization of temporal data.

described in section 7.2. We also assume an append-only storage model for the tuple time-stamping approach.

7.4.1 Comparison of Storage Consumption

In this section, we compare the storage consumptions of three approaches: instance, attribute and tuple time-stampings. The analytical formulas for the storage consumptions (in terms of physical pages) in these techniques are as follows (a step-by-step derivation of these formulas is given in Appendix D):

object instance time-stamping: $S_o = C*S*A*(TB*(N+2)+(M+ 2)*N_{avg})/Pg.$
tuple time-stamping: $S_t = C*S*A*(N+2)*N_{avg}/Pg.$
attribute time-stamping: $S_a = C*M*S*4*A*Navg/Pg.$

Examining the above formulas, it is obvious that parameters M, N_{avg} and N dictate the storage requirements in these approaches. As M increases, S_o and S_a increase. However, S_t is not affected by M because all attribute values of a historical version are stored in the tuple time-stamping approach no matter how many attributes are involved in an evolution of an object instance. As N increases, S_t increases proportionally. Since all attribute values of some snapshot databases are stored in the object instance time-stamping approach, S_o also increases as N increases. However, the increase in storage requirement in S_o is much less than that in S_t. Whereas S_a is not affected by N because only the changed attributes are stored in the attribute time-stamping approach. As N_{avg} increases, S_o, S_a and S_t all increase. Figures 7-14 to 7-17 show in detail how the storage requirement in each approach is affected by the parameters M (varied from 1 to 11), N (varied from 8 to 18 when M=3 and M=8, respectively) and N_{avg} (varied from 200 to 400 when M=3).

Figure 7-14 shows that the object instance time-stamping approach in general requires less storage space than the others when $1 < M < N$. This is because, in this

approach, only the changed attribute values are stored and time stamps are stored only once no matter how many attributes are modified at a time. However, if M is equal to 1, attribute time-stamping consumes the least amount of storage space because it does not require storage space for storing multi-snapshots. Intuitively, the attribute time-stamping approach consumes less storage space than the tuple time-stamping approach. However, this is not true if $M >= (N+2)/4$.

Figure 7-15 & 7-16 show that as N_{avg} increases (when M=3 and M=8, respectively), storage requirements in all three time-stamping techniques increase and the relative storage consumption depends on M: if M=1, $S_a > S_o > S_t$; if $1 < M < (N+2)/4$, $S_o > S_a > S_t$; otherwise, $S_o > S_t > S_a$.

Figure 7-17 shows that as N increases, the storage requirement of the tuple time-stamping approach increases proportionally; the storage requirement of the object instance time-stamping approach increases insignificantly (i.e., due to the storage for the snapshot in each TDB); the storage requirement of the attribute time-stamping approach is not affected.

7.4.2 Comparison of Computation Cost for Temporal Data Materialization

In this section, we present a comparison of computation costs for materializing temporal data using the three approaches. The computation cost in each technique consists mainly of the I/O time to access the temporal data, the CPU time to materialize all attribute values of an object instance (or tuple) and the I/O time to access and store intermediate results. We shall consider the materialization of temporal data of a time interval equivalent to a TDB duration and to the entire history during the comparison.

In the tuple time-stamping approach, since each historical record has the complete information, the computation cost consists of only the disk I/O time for

retrieving the temporal data which falls into the specified time interval. It is not necessary in this approach to perform any Select operation to retrieve the qualified data because the temporal data are sorted naturally in chronological order already.

In the attribute time-stamping approach, the computation cost consists of (1) the disk I/O time for retrieving those temporal tuples from each TNF relation which fall into the specified interval, (2) the disk I/O time for reading and writing the intermediate results from temporal join operations on those retrieved temporal data, and (3) the CPU time for joining the retrieved temporal data from the N-1 TNF relations into a N-attribute relation. Since the temporal data in each TNF relation are also sorted naturally in chronological order, it is not necessary in this approach to perform any Select operation to retrieve the qualified data. However, it would be very time-consuming to perform temporal join operations directly based on the way temporal data are organized in the secondary storage (i.e., the historical versions of one surrogate may not be stored in physically adjacent spaces) because several scans of a TNF relation would be required (e.g., using the nested-loop algorithm to perform a join). Therefore, it is more efficient to sort the temporal data of each TNF relation based on surrogates (or IIDs) and time tags in a chronological order before temporal join operations are performed. In our evaluation, we shall use a Temporal-Sort-Merge-Join algorithm (see appendix F for detail) which consists of a "two-phase sorting" [BLA77] algorithm and a "temporal join" algorithm to pre-sort and join TNF relations. By using this algorithm, the qualified data retrieved from one TNF relation will be sorted (based on surrogates and time tags) and stored in an intermediate relation. This intermediate relation will be temporally joined with the intermediate relation from the second TNF relation. Due to the fact that the time interval of a historical version of a surrogate in the first relation may not exactly match the time interval of a historical version of the surrogate in the

second relation, the time interval of each joined tuple needs to be aligned. As a result of the alignment, the join operation between two relations for each surrogate will produce a maximum of m+n and a minimum of max(m,n) tuples, where m is the number of historical versions of a surrogate involved in the first relation, n is the number of historical versions of the surrogate involved in the second relation, and max() is a function which returns the maximum number of its parameters (see Appendix F for detail). The intermediate result from joining the first two relations will be joined with the third relation, and etc. This process continues until all the N-1 intermediate relations are joined and the N-attribute relation is produced.

In the object instance time-stamping, since the temporal data in each TDB are clustered and sorted based on IIDs and in chronological order, the computational cost consists of (1) the disk I/O time for retrieving temporal data and (2) the CPU time for materializing historical object instance[8].

With nowadays technology, it is possible that the size of main memory of a system is large enough to accommodate all the intermediate results of joining the TNF relations in the attribute time-stamping approach. In this case, the "reads" and "writes" of all the intermediate results from the join operations for materializing temporal data can be avoided. The materialization cost of temporal data in attribute time-stamping thus consists of the disk I/O time for retrieving the temporal data and the CPU time for materializing the temporal data. Although the reads and writes of the intermediate results can be avoided when main memory is large, the attribute time-stamping approach still has the worst performance (except when M is less than or equal to two as shown in Figure 7-18) because temporal join operations are very costly. Figure 7-18 shows that

---

[8]In object instance time-stamping, since temporal data in each TDB are clustered and sorted based on IID and time order, materialization of temporal data can be completed in a single scan of the temporal data.

object instance time-stamping generally has the best performance in materializing temporal data and tuple time-stamping is better than attribute time-stamping as long as M is greater than two. Figures 7-19, 7-20, and 7-21 show the performances of the three approaches when N_{avg} is varied. The results in the three figures also confirm the conclusions in Figure 7-18: (1) when $M <= 2$, $C_o < C_a < C_t$[9] (Figures 7-19 and 7-20), and (2) when $M > 2$, $C_o < C_t < C_a$ (Figure 7-21).

However, if the main memory of a system is not large enough to hold all the intermediate results from joining TNF relations in the attribute time-stamping approach, the reads and writes for all the intermediate results need to be counted into the materialization cost. As a result, the above conclusion on computation costs has to be modified as follows. (1) The object instance time-stamping approach still has the best performance in general because the occupied storage space is much less than the others and it requires only a single scan of historical instances to materialize the needed temporal data of one TDB or even the entire history (see Figure 7-22). (2) The attribute time-stamping approach has a much worse performance than the other two approaches regardless of the value of M because it requires temporal join operations for the TNF relations and "reads" and "writes" for the intermediate results from these operations. That is, $C_o < C_t < C_a$ (Figures 7-22, 7-23, 7-24, and 7-25). (3) As N_{avg} increases, the materialization costs in C_t, C_o, and C_a all increase (Figures 7-24 and 25). A detailed analysis and the formulas for calculating the computation costs of the three approaches when I/Os of the intermediate results are included in the attribute time-stamping approach are given in the Appendix F.

---

[9] C_a: computation cost for materializing temporal data in attribute time-stamping. C_o: computation cost of materializing temporal data in object instance time-stamping. C_t: computation cost for materializing temporal data in tuple time-stamping.

7.4.3 Processing Costs of Benchmark Temporal Queries

The processing cost of a temporal query consists of the cost of searching data indices, the cost of retrieving temporal data from secondary storage to main memory, and the cost of materializing temporal data in main memory. However, in systems using the same storage model but different indexing techniques, the processing costs of temporal queries can be determined based on the performance of index processing.

In this section, we shall compare our layered index mechanism with those indexing techniques described in Section 7.2 based on the processing costs of benchmark temporal queries. Since the data of a N-attribute tuple in the attribute time-stamping approach are scattered into N-1 TNF relations, it is necessary and expensive to search N-1 indices, one for each of the TNF relations, to process the data for a temporal query. Besides, the attribute time-stamping approach has a much worse performance in most cases than the others in materializing temporal data as shown in the preceding section. It is obvious that its performance using an indexing technique will also be worse than the other time-stamping approaches using the same indexing technique. Therefore, in this section, we will compare our DIMS storage model and the layered index mechanism against only the tuple time-stamping approach and the various existing indexing techniques. The comparison for processing temporal queries are based on the costs of searching indices and retrieving temporal data. In this comparison, we also make the following assumptions for both storage models: (1) for query types 1 through 7, we assume that an index on IID is available; for query types 8 to 10, we assume that an index on attribute values is available, and (2) current data are separated from historical data. For the convenience of naming different techniques, we assign each index technique and its underlying storage model with a symbol as follows.

I_1: The indexing scheme of the proposed DIMS storage model,
I_2: The accession list [AHN88],

I_3: The Time Index [ELM90a],
I_4: The two-level Attribute/Time Index [ELM90a],
I_5: The ST-tree (or AT-tree) [GUN90].

The processing cost for each indexing technique and its associated storage model is expressed in terms of disk I/Os in our analytical formulas which are given in Appendix G.

In our analysis, the values of the parameters in the formulas are varied in the ranges shown in Appendix E. We shall summarize below the results of a performance evaluation based on the effects of parameter changes to different groups of benchmark query types. A more detailed description of the results is given in Appendix H.

7.4.3.1 Performance evaluation for query types 1 to 5:

The performances of indexing techniques for query types 1 to 5 are sensitive to the value changes of the parameters M, N_{avg}, Pg, n, S, and S_{avg}. They are shown in Table 7-2 through Table 7-5. Table 7-2 shows the relative performance when M is varied. It shows that I_1 has the best performance for query types 1, 2 and 4 when M is less than 22. This is because all the required temporal data in the DIMS storage model are aggregated together in a physical block and the required number of disk I/Os is therefore greatly reduced. However, if M increases to 37 or larger, the number of disk I/Os increases and the performance of I_1 degrades. Nevertheless, the relative performance among I_2 to I_5 stays constant since these index techniques are not affected by M. I_2 has the best performance for query type 5 because all the historical versions of an object instance can be retrieved in a block manner once the current version of the object instance is located. I_3 is the best for query type 3 because the pointers of all the required data at a time point are aggregated together in a bucket.

Table 7-3 through Table 7-5 show the relative performances of these indexing techniques when the average number of evolutions of object instances N_{avg}, the page size

Pg, and the number of data blocks n involved in a time interval are varied. Generally speaking, their performances are similar to the results, shown in Table 7-2. However, if these parameters exceed some threshold values, their relative performances change. For example, in query type 1, I_4 and I_5 have better performance than I_2 when N_{avg} exceeds 600 and than I_1 when N_{avg} exceeds 1300.

The total number of object instances S also affects the performance of an indexing technique because, as S increases, the number of disk I/Os in each indexing technique increases. Since S appears in a logarithmic function in the analytical formulas of all the indexing techniques, the increase in disk I/Os is logarithmically proportional to the increase of S in all techniques. Therefore, their relative performances do not change when S is varied.

7.4.3.2 Performance evaluation for query types 6 to 7:

Query type 6 retrieves the current version of an object instance. Since the data of current version is physically separated from historical data in our assumption, the performances of I_1 to I_5 are affected by the sizes of index trees (i.e.,Pg and S). In this query type, I_1 and I_2 have the best performance because they have an index on IIDs of current instances. The performances of I_4 and I_5 are very close to that of I_1 and I_2. I_3 has the worst performance because of the sequential access of the bucket of pointers to current data.

Query type 7 retrieves all the current versions of object instances. In processing this query type, a sequential search over the current data area are performed for all indexing techniques. The cost of this query type is identical for all techniques and is represented by the expression "1 + ceil(S*(N+2)*A/Pg)", where 1 accounts for the disk I/O to access the beginning address of the current data area and ceil(S*(N+2)*A/Pg)

accounts for the number of disk I/Os to sequentially read all the current versions of the object instances.

7.4.3.3 Performance evaluation for query types 8 to 10:

For query types 8 to 10, there are a number of parameters which are common and important to the processing and searching of every indexing technique. They are the number of distinct values of an indexed attribute in the database at a given time, the number of evolutions[10] of each of the distinct attribute values in the database through the sampled time period, and the number of object instances which have the same attribute values at each time point. For the evaluation of query types 8 to 10, we assume that (1) the total number of distinct values of an indexed attribute in the database at a given time is NA, (2) each of the NA values has the same number of evolutions EV which are uniformly distributed along the time dimension where $EV=(S*N_{avg}*M/((N-1)*NA))$ and $(S*N_{avg}*M/(N-1))$ is the total number of evolutions for one attribute of an object class, and (3) the number of object instances which have the same attribute value is $k=S/NA$. The performances of indexing techniques for query types 8 to 10 are sensitive to the value changes of the parameters M, N_{avg}, Pg, n, S, NA and k.

For query type 8, it is obvious that I_2 has the best performance because it indexes only on current data and the search for the qualified current data can be completed without traversing through the history chains. I_1 performs better than I_4 and I_5 by $\log_f(TB)$. If $\log_f(TB)$ is insignificant when compared with the other parameters (such as when $N_{avg}<600$, $M>=2$, Pg $>=$ 2K bytes, or $S<20000$), the performances of I_1, I_4 and I_5 are identical. I_3 has the worst performance in this query type because it has to search all

[10]In general, one attribute value AV will be associated with a set of object instances (i.e., all the object instances in the set have the attribute value AV) for a period of time. This set of object instances, however, may be changed (i.e., the attribute of object instances are changed to and from AV) as the database evolves. When such a change occurs, it is counted as an evolution of the attribute value AV.

the current data sequentially. However, if the selectivity parameter k increases, the "false hit" rate in I_3 decreases and the performance of I_3 approaches to the others and becomes the best when k is equal to S.

For query type 9, I_4 and I_5 have the best performance because only the qualified data will be accessed. I_2 has the worst performance because searching temporal data in I_2 always has to start from the current data and the traversals through the Accession Lists for all the object instances are expensive. The performance of I_1 is worse than those of I_4 and I_5 because traversals through the historical chains are required in I_1. However, in cases when the required data in I_1 are available directly from the snapshot, I_1 will have better performance than I_4 and I_5 because the traversals of historical chains in I_1 is avoided. I_3 is only better than I_2 because, in I_3, it is required to check all the temporal data of the interested time point. However, since I_3 is not affected by the increase of k while the other indexing techniques tend to require more disk I/Os, it performs relatively better than I1 when k > = 4000 and becomes the best when k is equal to S.

For query type 10, I_4 and I_5 have the best performance because only the qualified data will be accessed. I_3 has the worst performance because it has to search through all the temporal data of the specified time interval no matter whether these data satisfy the attribute predicate or not. I_2 is worse than I_1 because searching temporal data in I_2 always has to start from the current data and the expensive traversals through the Accession Lists for all the object instances are necessary. I_1 is worse than I_4 and I_5 because of the need of traversing the historical chains. The relative performances of indexing techniques, i.e., $I_4 = I_5 < I_1 < I_2 < I_3$, are not affected by the value change of any parameter in this query type.

Based on our analysis, the performance of an indexing technique and its storage model can be determined by the following parameters: M, N_{avg}, Pg, n, S, S_{avg}, NA, and k. Since the temporal data in the storage models for I_2, I_3, I_4 and I_5 are non-compressed, the parameter M does not affect them. It only affects I_1 since, as M increases, the number of disk I/Os in I_1 also increases and the performance of I_1 degrades. The parameters N_{avg}, Pg and S affects all the indexing techniques. As N_{avg} and S increase, the amount of temporal data of a time interval increases and so does the required number of disk I/Os for processing these temporal data. As Pg increases, the fan-out factor f for an index tree increases and thus the required number of disk I/Os for searching the index decreases. The number of retrieved data blocks n involved in a time interval also affects the performance of an indexing technique. As n increases, the amount of temporal data to be processed also increases and so does the required number of disk I/Os. However, the increase in the number of disk I/Os in I_1 is more than those in the other indexing techniques because the system will have to search a local index tree in I_1 for each of the n TDBs. S_{avg} only affects I_3. The number of distinct attribute values NA and the selectivity factor k affect the performances of the indexing techniques only in query types 8 to 10. As NA increases, the selectivity decreases and so dose the number of required disk I/Os in each indexing technique. As k increases, the selectivity increases and so does the number of required disk I/Os. I3, however, is not affected by NA and k for query types 9 and 10.

7.4.4 Conclusions on Performance Analysis and Comparison

We have analyzed and compared the performance of our proposed storage model and layered index mechanism with a number of existing index techniques in terms of

storage consumption, cost of temporal data materialization, and processing costs of a set of benchmark queries. We conclude our analysis and comparison below:

For storage consumption, the DIMS storage model consumes the least amount of storage space as long as M is greater than one and less than N. Attribute time-stamping using TNF relations consumes less storage space than tuple time-stamping when $M < (N+2)/4$; otherwise, tuple time-stamping is better.

In materializing temporal data of a time interval or the entire history, the performances of different storage models depend on the number of disk I/Os required to retrieve the temporal data from secondary storage, the number of disk I/Os required to read and write the intermediate results, and the computation time needed to materialize a complete record. Since the DIMS storage model occupies the least amount of storage space and requires only a single scan of the data files to materialize the temporal data, it has the best performance. The tuple time-stamping approach has better performance than the attribute time-stamping approach in most cases because, in the attribute time-stamping approach, the expensive temporal join operations and the reads and writes for the intermediate results from joining TNF relations are required. The attribute time-stamping approach is only better than the tuple time-stamping approach when $M <= 2$ under the assumption that the main memory is large enough to accommodate all the intermediate results from joining the TNF relations.

In processing the ten types of benchmark temporal queries, the layered indexing mechanism of the DIMS storage model is better than the other techniques in query types 1, 2, 4, and 6. This is because temporal data in the DIMS model are compressed and stored in a smaller number of adjacent physical blocks. For query type 7, all the indexing techniques have the same performance if we assume that the current data are physically separated from the historical data; otherwise, $I_1 = I_2 < I_3 < I_4 = I_5$. Although

I_1 does not have the best performance in some query types due to the overhead in traversing the history chains and in retrieving the snapshot data which are physically separated from the delta instances in each TDB, it is quite comparable to the other indexing techniques. For example, I_1 is only inferior to I_3 in query type 3, and is very close to I_2 which is the best for query type 8. I_3 has the best performance for query type 3 because all the pointers of the required temporal data are gathered in a bucket. It is also the best for query types 8 and 9 under the condition when k is equal to S. I_2 is the best for query types 5 and 8 because it indexes only on current data and the search for the qualified data can be completed without traversing through the history chains. I_4 and I_5 have the best performance for query types 9 and 10 under the assumption that the selectivity is identical at each time point and the number of qualified object instances is equal to k (=S/NA). This is because, under the assumption, the rate of "false hit" is greatly reduced and only the qualified data and their pointers will be retrieved. If the assumption does not hold, the performances of I4 and I5 will deteriorate.

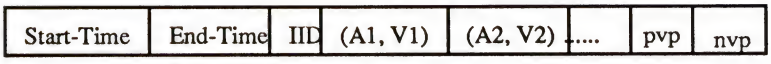


Figure 7-1: Data format of a temporal object instance.

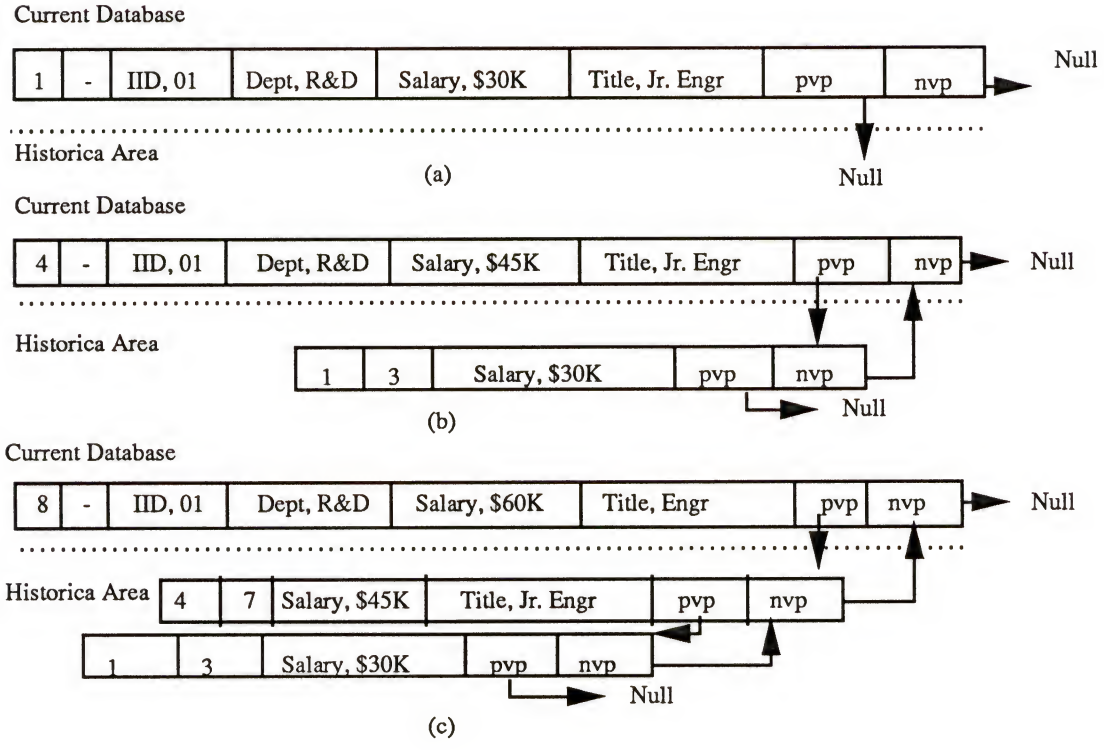


Figure 7-2: Delta-Instance concept for compressing temporal object instances. a) When John's data was created at time 1; b) When John's data was modified at time 4; b) When John's data was modified at time 8.

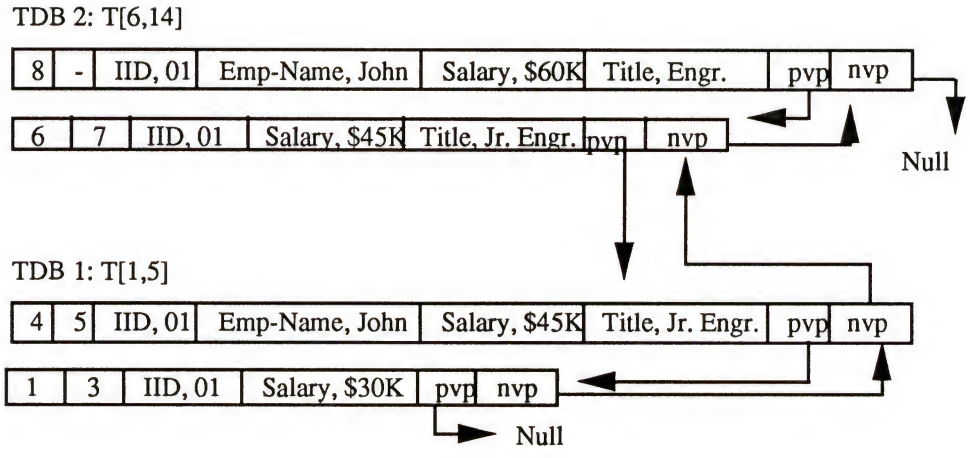


Figure 7-3: Partitioning the temporal data of employee John based on T[1,5], T[6,14], etc.

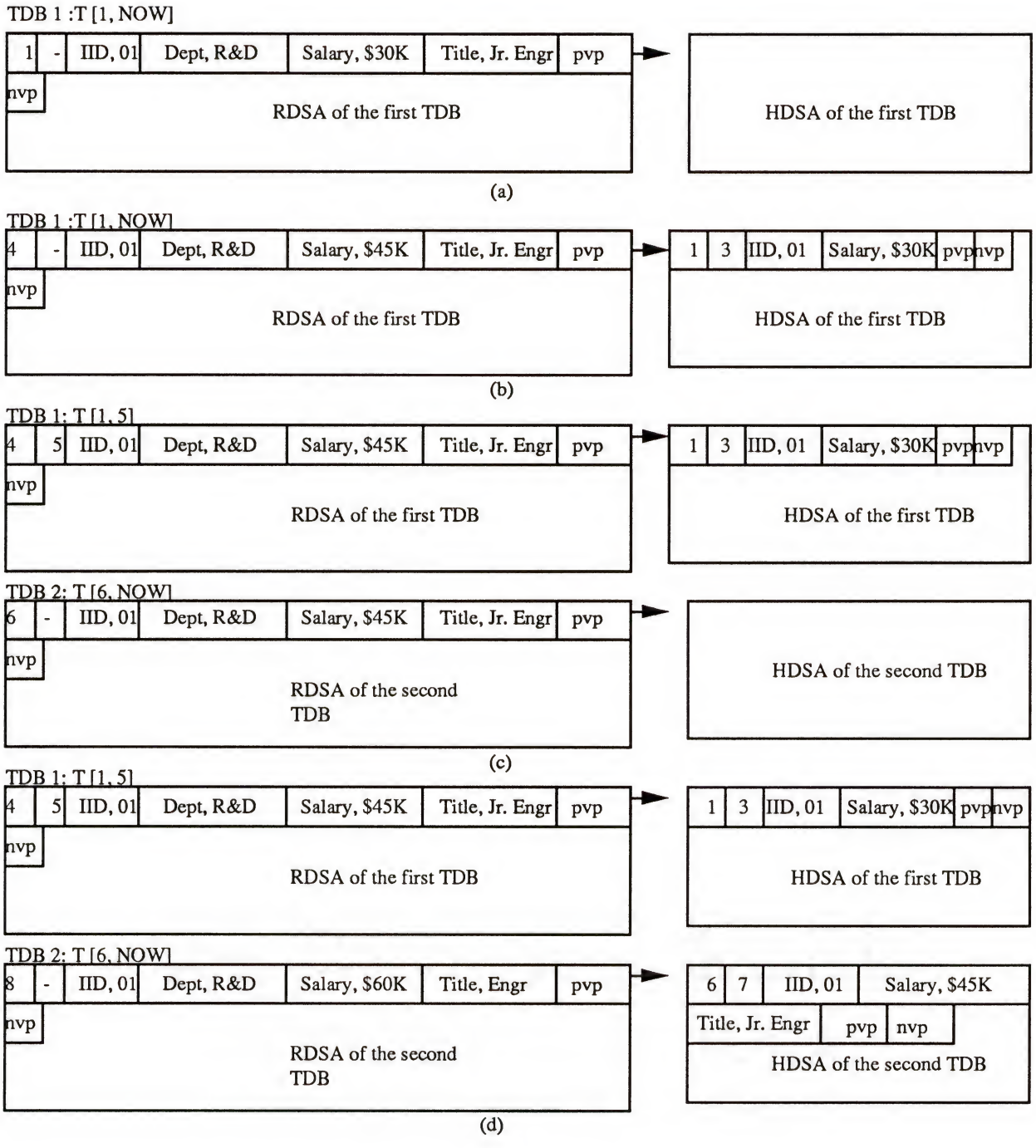


Figure 7-4: The formation of TDB's.
a) When the object instance John was initially created at time 1; b) When John was modified at time 4; c) When the first TDB expires (i.e., at time 6); d) When John was modified at time 8.

Start Time	End Time	SSN	Salary	Addres	....	nvp
1	4	0001	$20K	a1		
2	3	0002	$30K	a3		
2	7	0003	$28K	a4		
4	9	0002	$35K	a3		
5	5	0001	$32K	a1		
6	-	0001	$32K	a7		
10	-	0002	$35K	a6		
. .	. .	. .	. .	. .		

Figure 7-5: The storage model using tuple time-stamping.

TNF 1

Start Time	End Time	SSN	Salary	nvp
1	4	0001	$20K	
2	3	0002	$30K	
2	7	0003	$28K	
4	-	0002	$35K	
5	-	0001	$32K	
. .	. .	. .	. .	

TNF 2

Start Time	End Time	SSN	Addres	nvp
1	5	0001	a1	
2	9	0002	a3	
2	7	0003	a4	
6	-	0001	a7	
10	-	0002	a6	
. .	. .	. .	. .	

Figure 7-6: The storage model using attribute time-stamping.

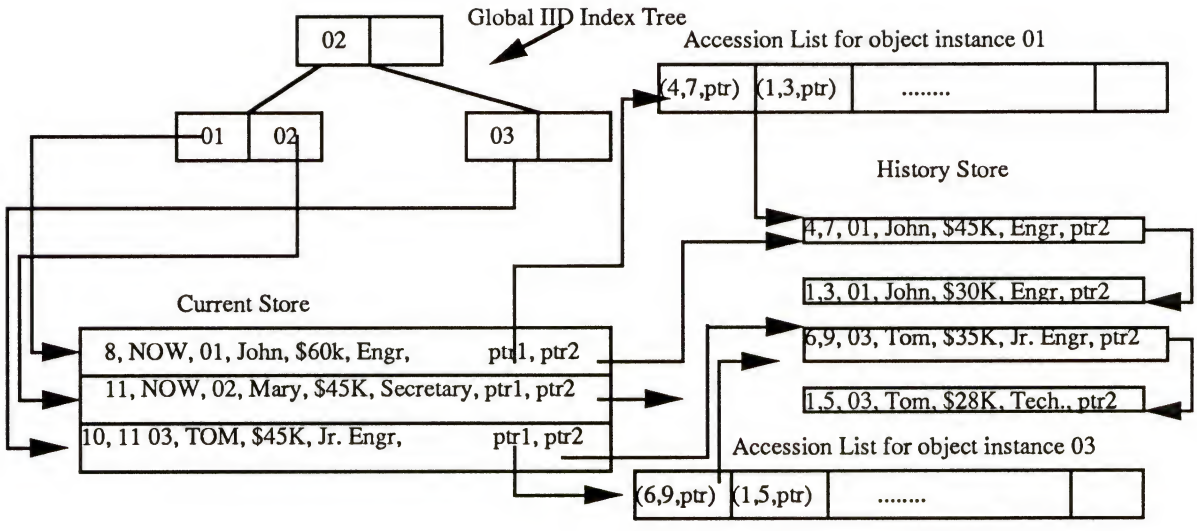


Figure 7-7: Illustration of Accession List.

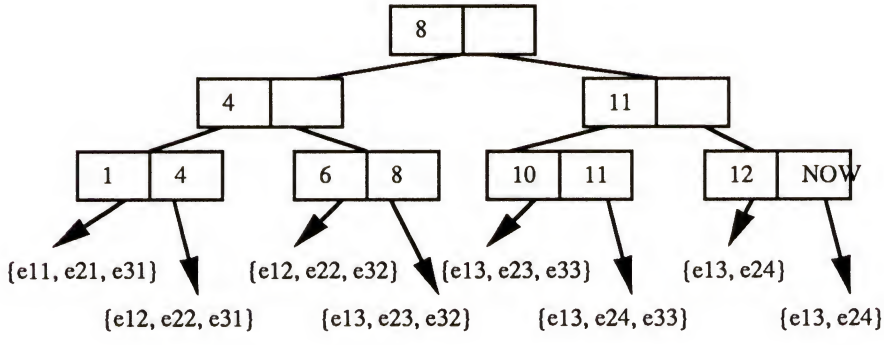


Figure 7-8: Illustration of Time Index.

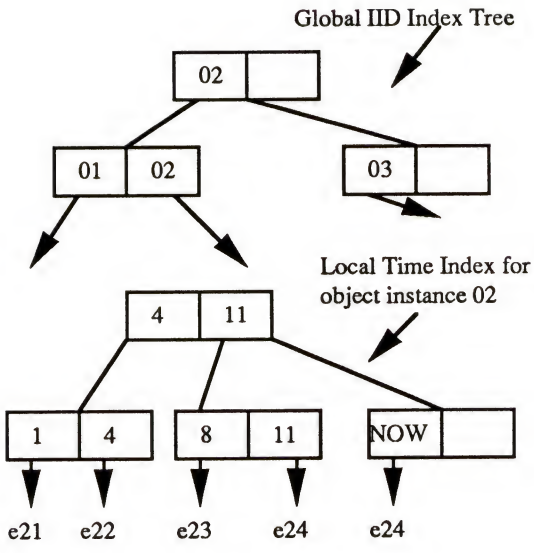


Figure 7-9: Two-level Time Index with first-level index on IID's.

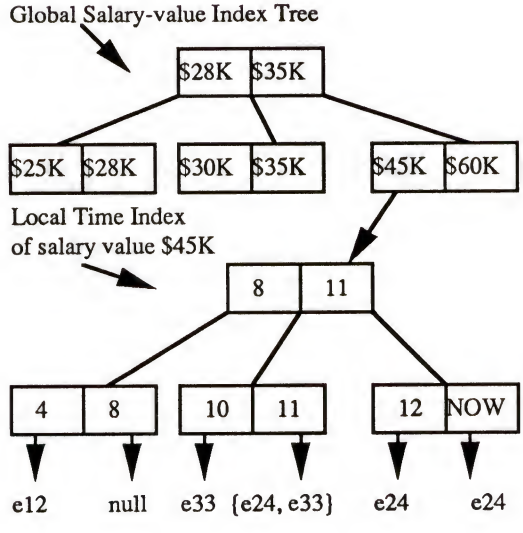


Figure 7-10: Two-level Time Index with first-level index on Salary values.

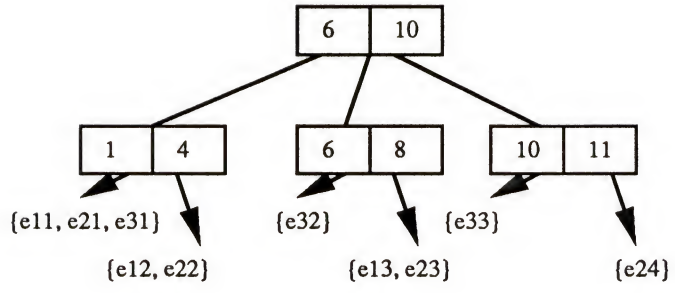


Figure 7-11: Illustration of AP-tree.

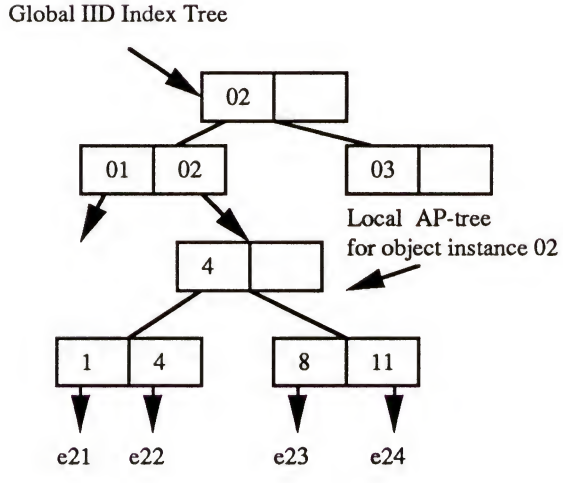


Figure 7-12: Two-level ST-tree with first-level index on IID's.

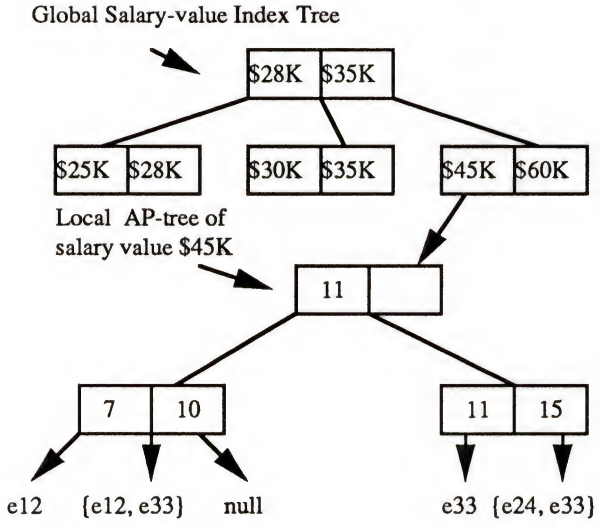


Figure 7-13: Two-level AT-tree with first-level index on Salary values.

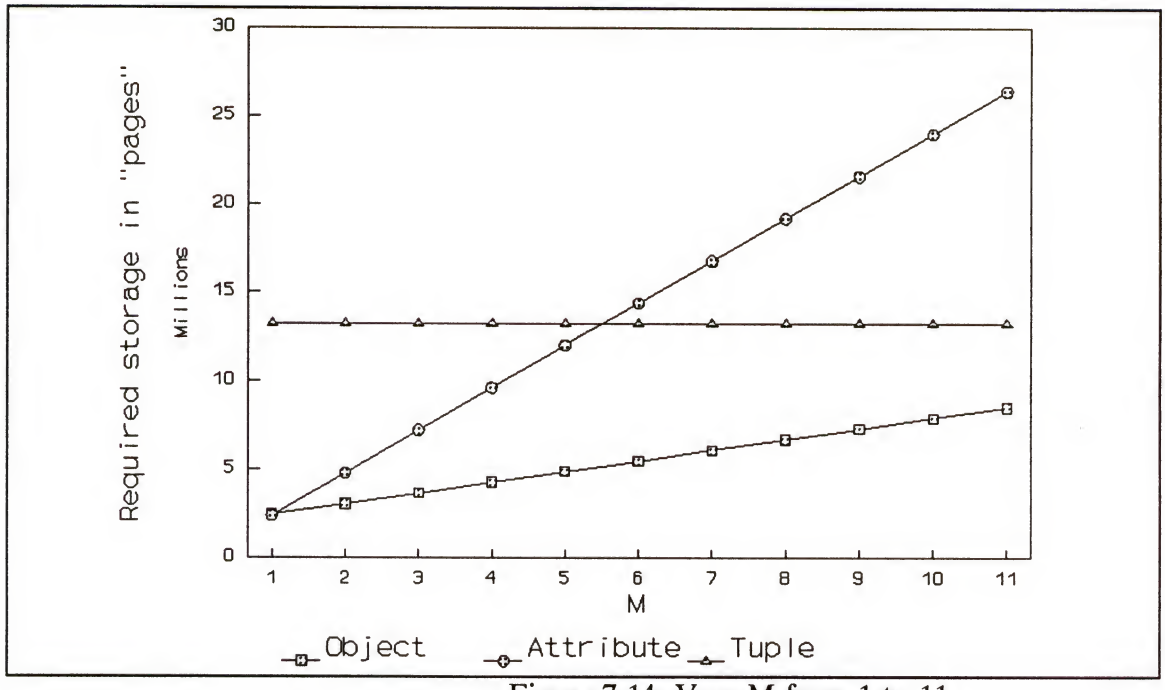


Figure 7-14: Vary M from 1 to 11.

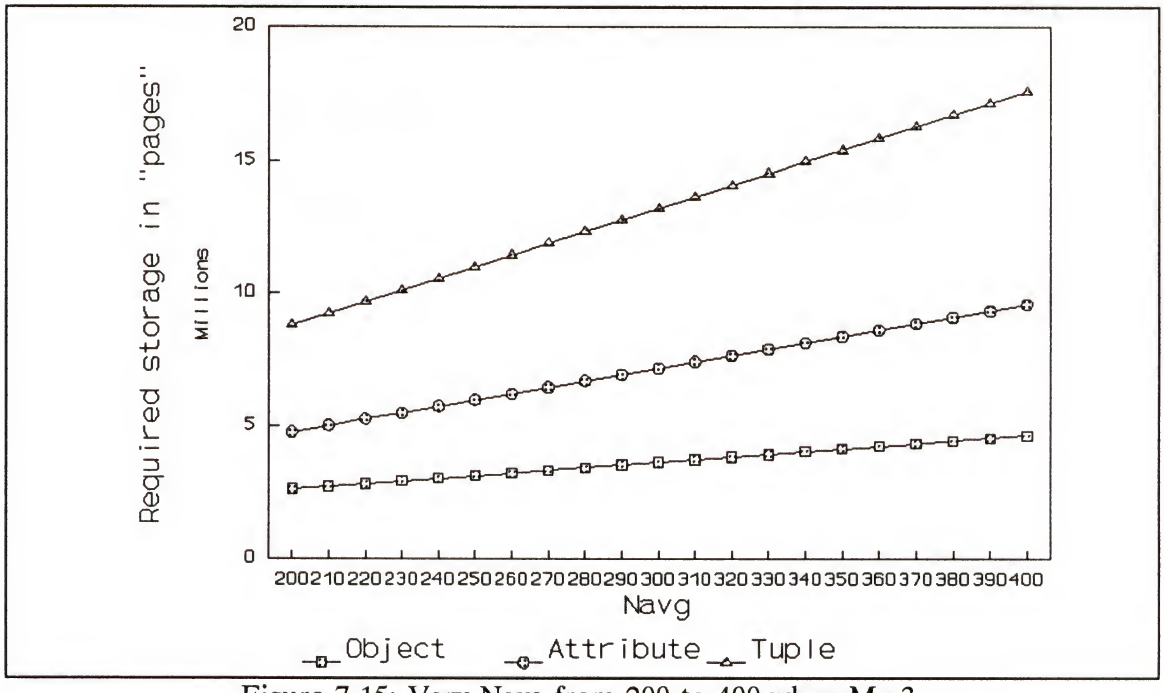


Figure 7-15: Vary Navg from 200 to 400 when M=3.

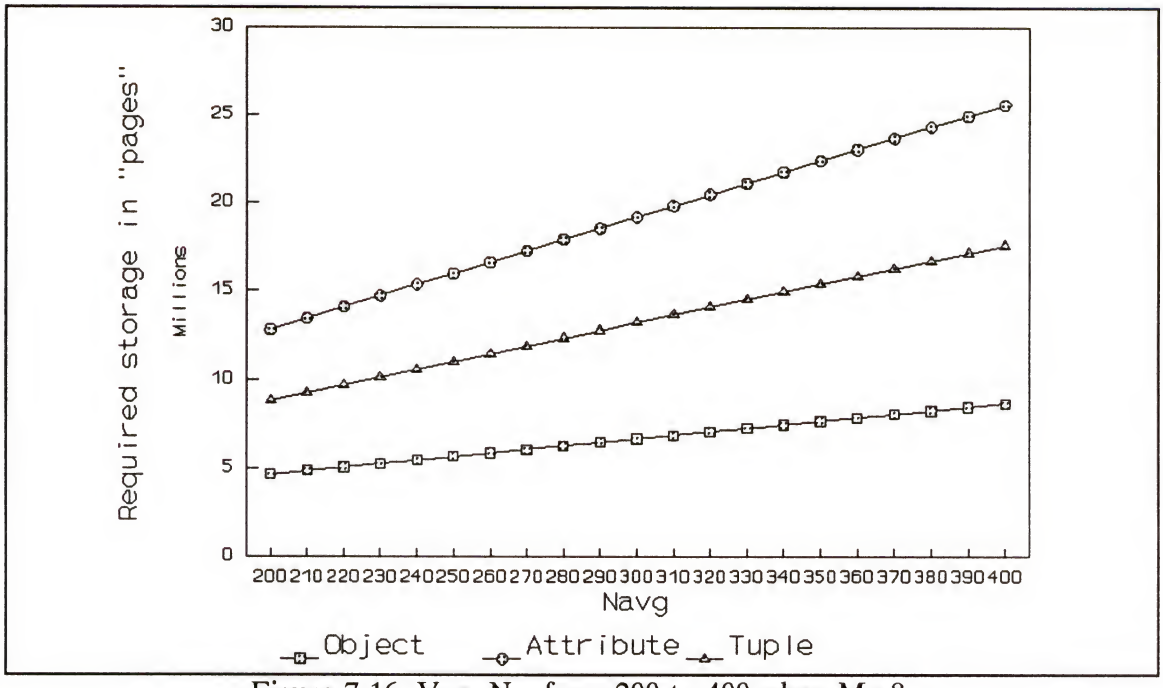


Figure 7-16: Vary N_{avg} from 200 to 400 when M=8.

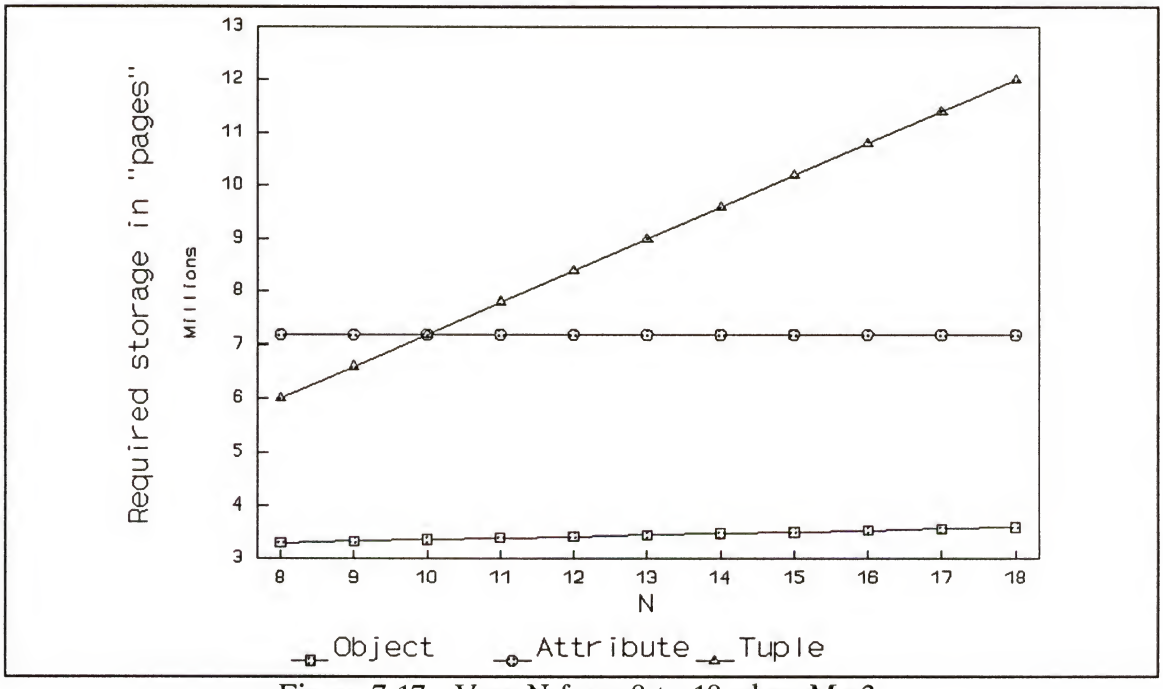


Figure 7-17 : Vary N from 8 to 18 when M=3.

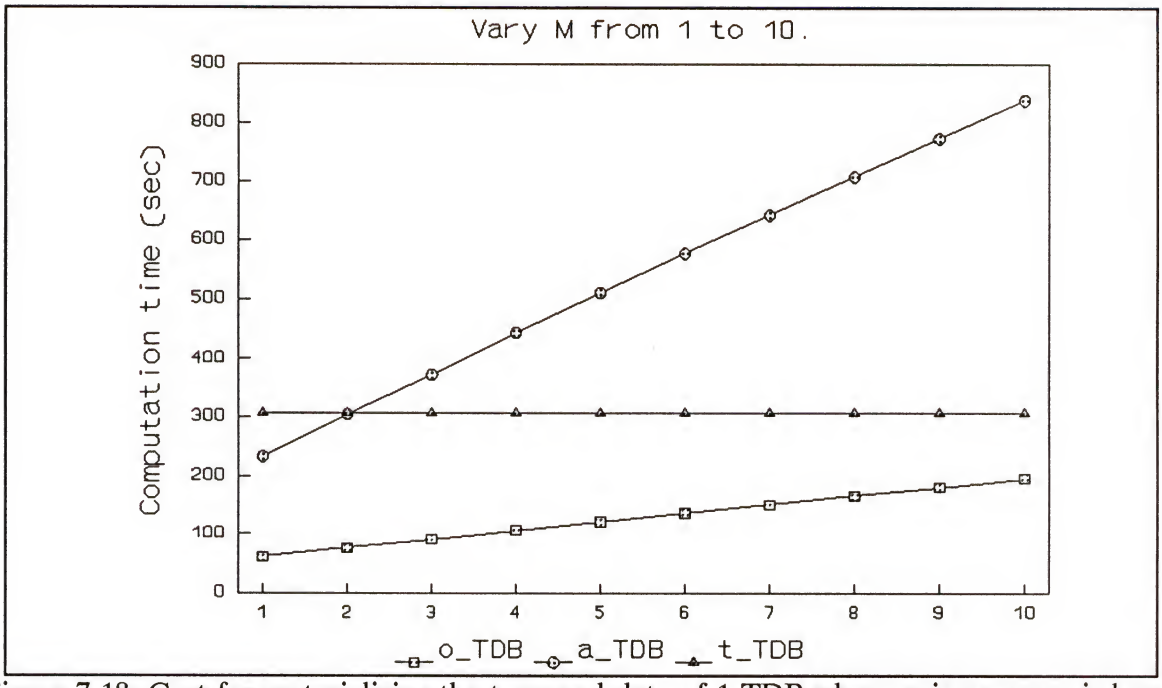


Figure 7-18: Cost for materializing the temporal data of 1 TDB when main memory is large.

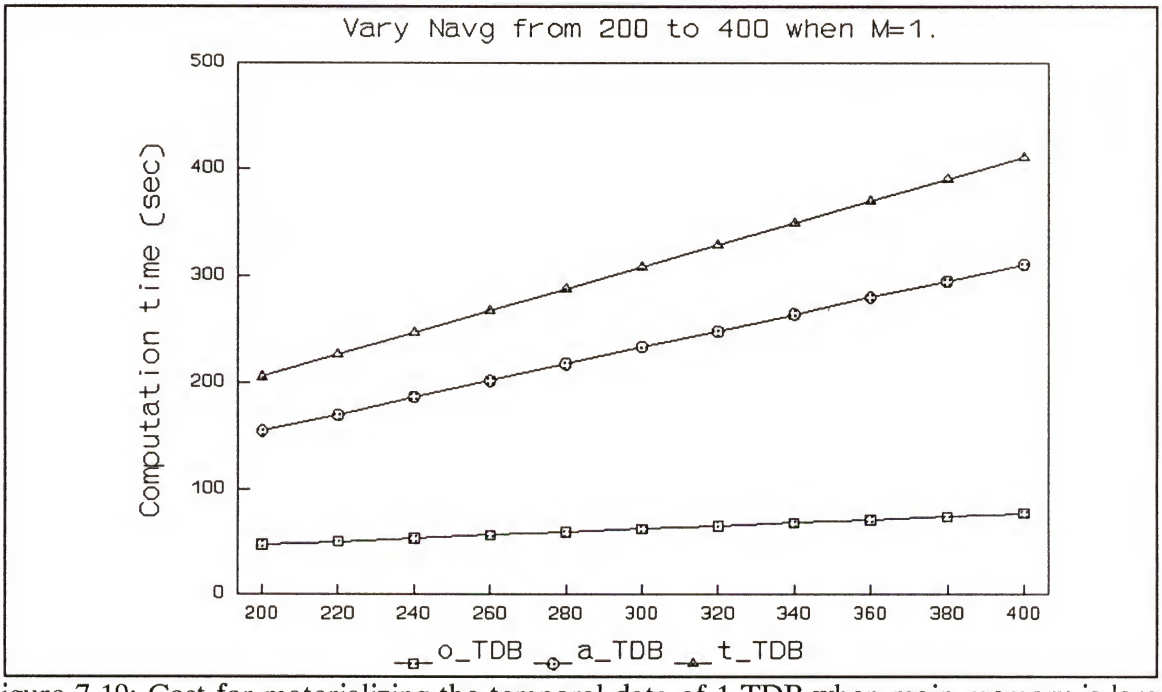


Figure 7-19: Cost for materializing the temporal data of 1 TDB when main memory is large.

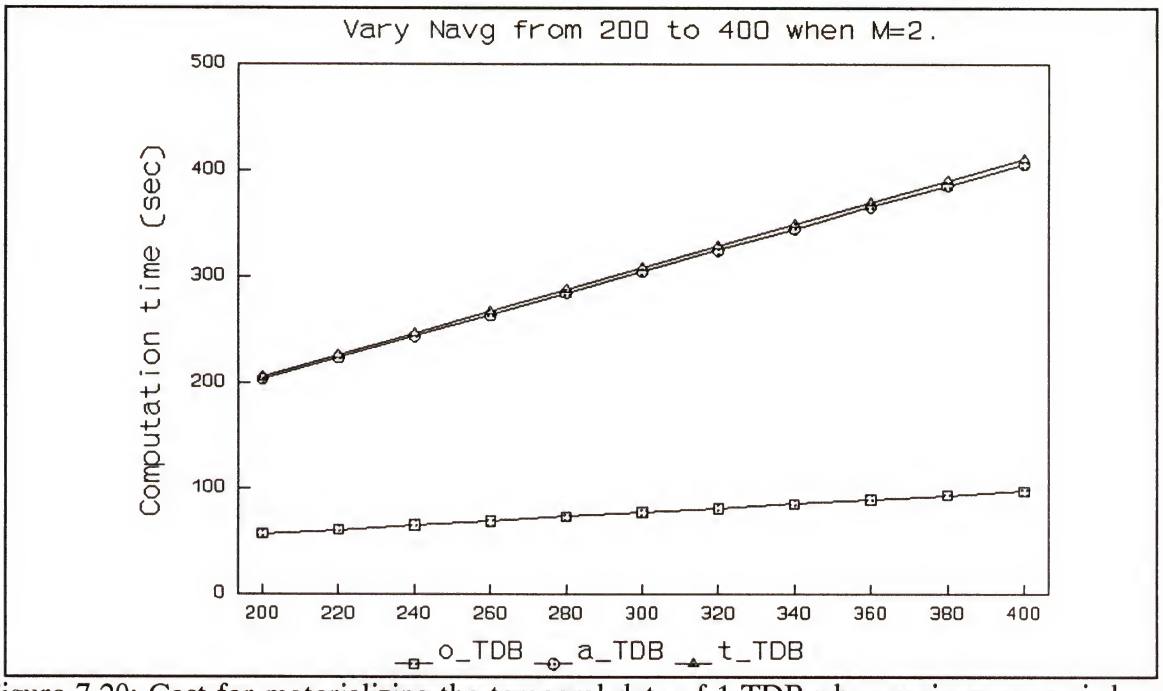


Figure 7-20: Cost for materializing the temporal data of 1 TDB when main memory is large.

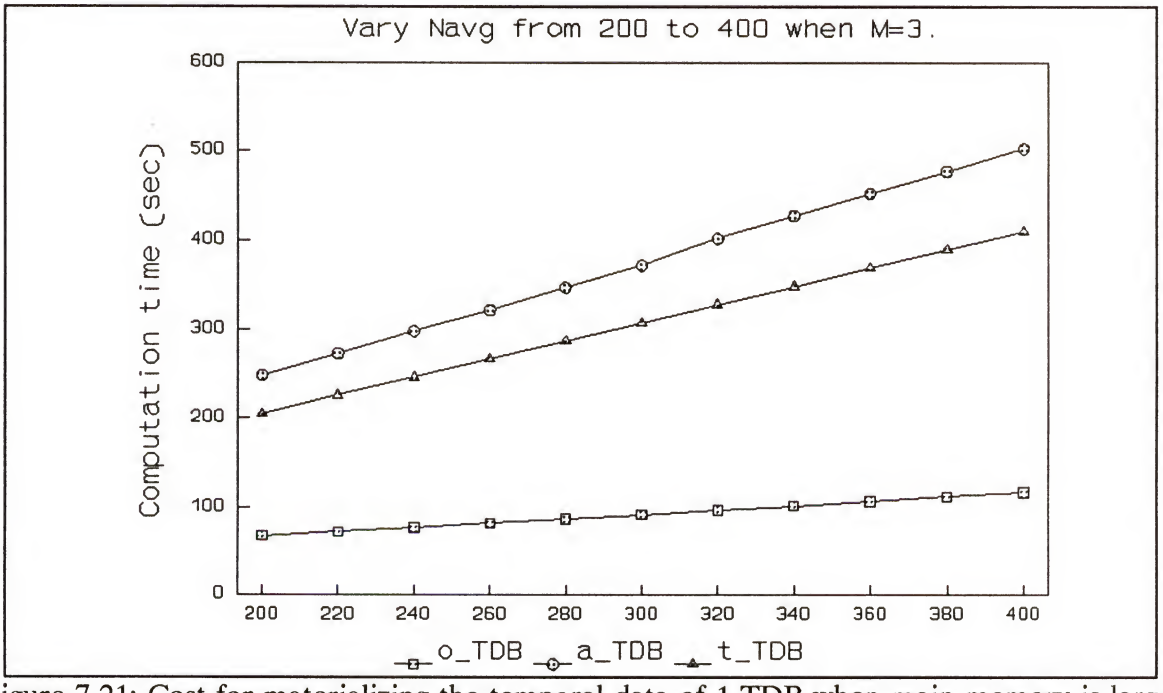


Figure 7-21: Cost for materializing the temporal data of 1 TDB when main memory is large.

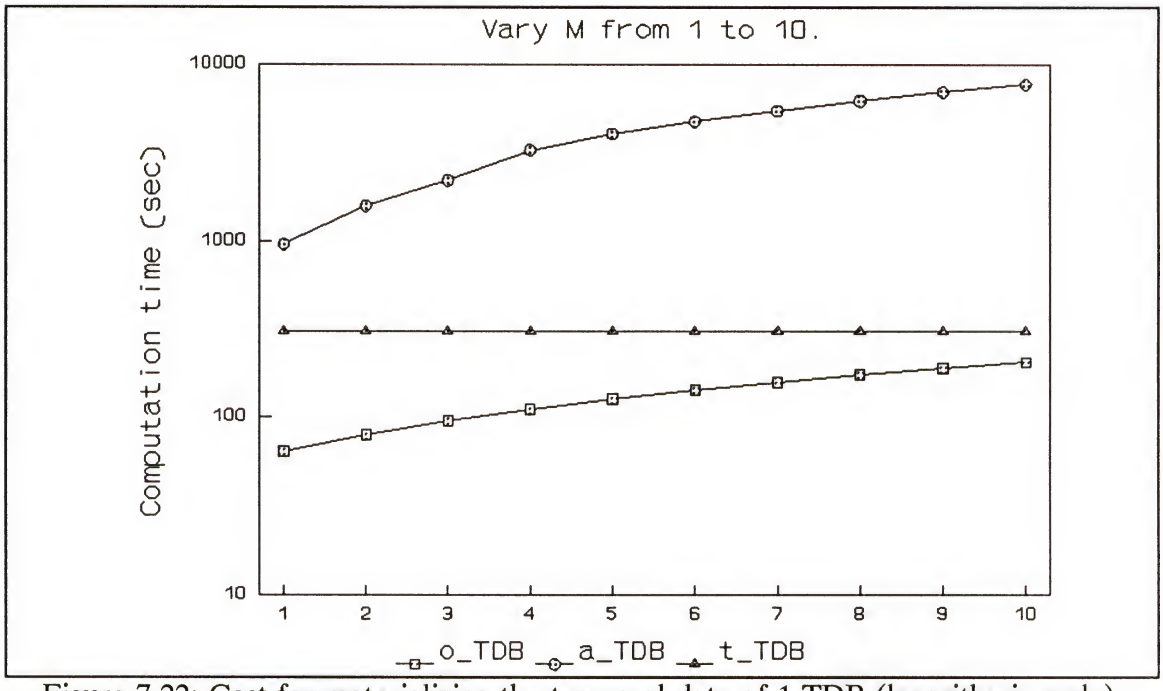


Figure 7-22: Cost for materializing the temporal data of 1 TDB (logarithmic scale).

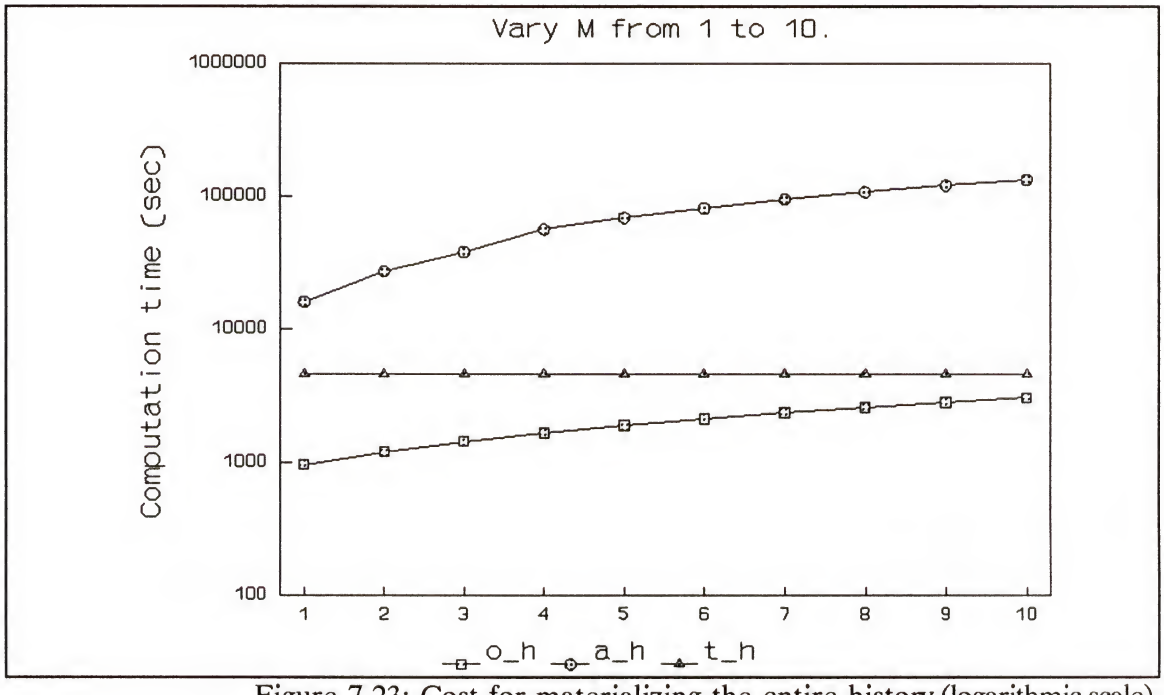


Figure 7-23: Cost for materializing the entire history (logarithmic scale).

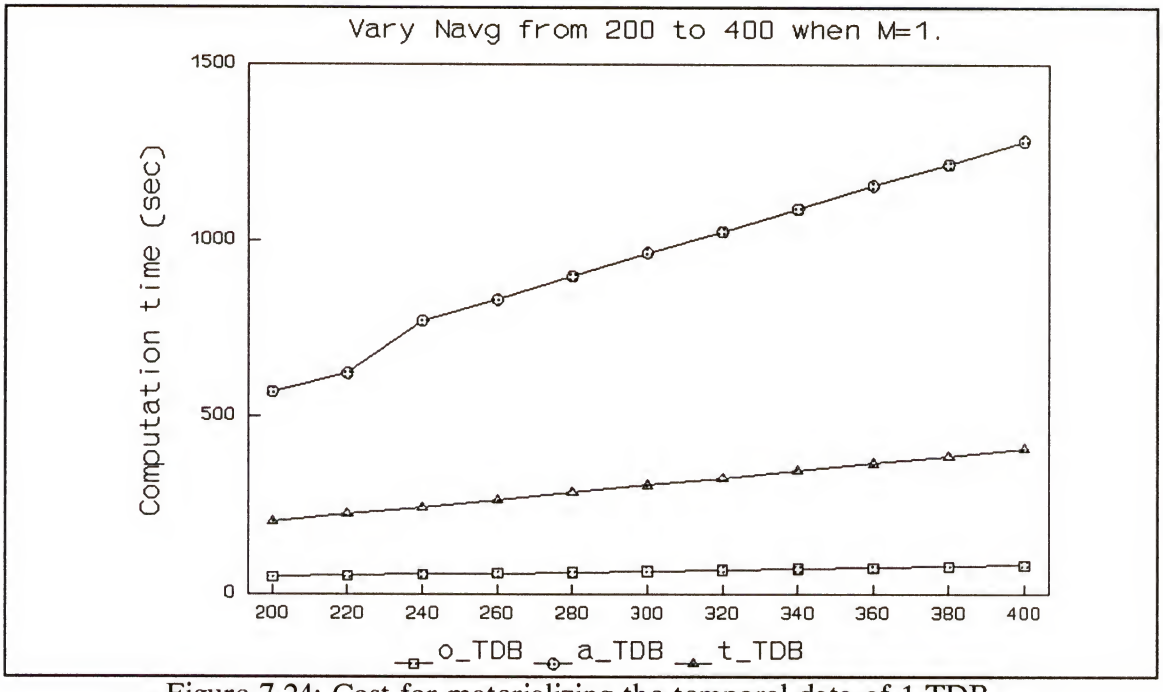


Figure 7-24: Cost for materializing the temporal data of 1 TDB.

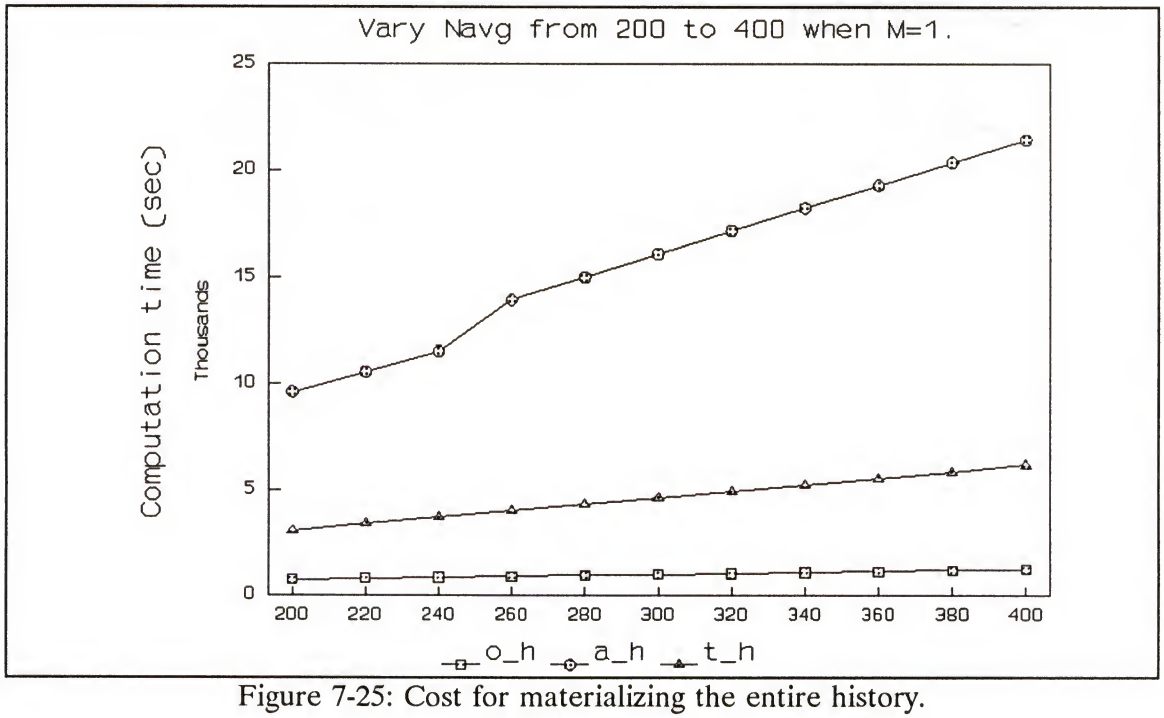


Figure 7-25: Cost for materializing the entire history.

Table 7-1: Temporal data of Employee class.

TIID	Start time	End time	IID	Emp Name	Salary	Title
e_{13}	8	-	01	John	$60K	Engr
e_{12}	4	7	01	John	$45K	Jr. Engr
e_{11}	1	3	01	John	$30K	Jr. Engr
e_{24}	11	-	02	Mary	$45K	Secretary
e_{23}	8	10	02	Mary	$30K	Secretary
e_{22}	4	7	02	Mary	$30K	Clerk
e_{21}	1	3	02	Mary	$25K	Clerk
e_{33}	10	11	03	Tom	$45K	Jr. Engr.
e_{32}	6	9	03	Tom	$35K	Jr. Engr.
e_{31}	1	5	03	Tom	$28K	Technician

Table 7-2: The relative performances when M is varied.

	$M<22$	$22<=M<23$	$23<=M<25$	$25<=M<33$
Q1	$I_1<I_2<I_4=I_5<I_3$	$I_1<I_2<I_4=I_5<I_3$	$I_1<I_2<I_4=I_5<I_3$	$I_1=I_2<I_4=I_5<I_3$
Q2	$I_1<I_2<I_4=I_5<I_3$	$I_1<I_2<I_4=I_5<I_3$	$I_2<I_1<I_4=I_5<I_3$	$I_2<I_1=I_4=I_5<I_3$
Q3	$I_3<I_1<I_2<I_4=I_5$	$I_3<I_2<I_1<I_4=I_5$	$I_3<I_2<I_1<I_4=I_5$	$I_3<I_2<I_1<I_4=I_5$
Q4	$I_1<I_3<I_2<I_4=I_5$	$I_1<I_3<I_2<I_4=I_5$	$I_1<I_3<I_2<I_4=I_5$	$I_1<I_3<I_2<I_4=I_5$
Q5	$I_2<I_4=I_5<I_1<I_3$	$I_2<I_4=I_5<I_1<I_3$	$I_2<I_4=I_5<I_1<I_3$	$I_2<I_4=I_5<I_3<I_1$

Table 7-2: continued.

	$33<=M<34$	$34<=M<37$	$37<=M<700$	$700<=M<720$
Q1	$I_1=I_2<I_4=I_5<I_3$	$I_1=I_2<I_4=I_5<I_3$	$I_2<I_4=I_5<I_1<I_3$	$I_2<I_4=I_5<I_3<I_1$
Q2	$I_2<I_4=I_5<I_1<I_3$	$I_2<I_4=I_5<I_1<I_3$	$I_2<I_4=I_5<I_1<I_3$	$I_2<I_4=I_5<I_1<I_3$
Q3	$I_3<I_2<I_1<I_4=I_5$	$I_3<I_2<I_4=I_5<I_1$	$I_3<I_2<I_4=I_5<I_1$	$I_3<I_2<I_4=I_5<I_1$
Q4	$I_1<I_3<I_2<I_4=I_5$	$I_1<I_3<I_2<I_4=I_5$	$I_1<I_3<I_2<I_4=I_5$	$I_1<I_3<I_2<I_4=I_5$
Q5	$I_2<I_4=I_5<I_3<I_1$	$I_2<I_4=I_5<I_3<I_1$	$I_2<I_4=I_5<I_3<I_1$	$I_2<I_4=I_5<I_3<I_1$

Table 7-2: continued.

	$720<=M<1900$	$1900<=M<2000$	$2000<=M$
Q1	$I_2<I_4=I_5<I_3<I_1$	$I_2<I_4=I_5<I_3<I_1$	$I_2<I_4=I_5<I_3<I_1$
Q2	$I_2<I_4=I_5<I_3<I_1$	$I_2<I_4=I_5<I_3<I_1$	$I_2<I_4=I_5<I_3<I_1$
Q3	$I_3<I_2<I_4=I_5<I_1$	$I_3<I_2<I_4=I_5<I_1$	$I_3<I_2<I_4=I_5<I_1$
Q4	$I_1<I_3<I_2<I_4=I_5$	$I_3<I_1<I_2<I_4=I_5$	$I_3<I_2<I_4=I_5<I_1$
Q5	$I_2<I_4=I_5<I_3<I_1$	$I_2<I_4=I_5<I_3<I_1$	$I_2<I_4=I_5<I_3<I_1$

Table 7-3: The relative performances when N_{avg} is varied.

	$N_{avg} < 100$	$100 <= N_{avg} < 400$	$400 <= N_{avg} < 500$	$500 <= N_{avg} < 520$
Q1	$I_1 < I_2 < I_4 = I_5 < I_3$	$I_1 < I_2 < I_4 = I_5 < I_3$	$I_1 < I_2 < I_4 = I_5 < I_3$	$I_1 < I_2 < I_4 = I_5 < I_3$
Q2	$I_1 = I_2 = I_4 = I_5 < I_3$	$I_1 < I_2 < I_4 = I_5 < I_3$	$I_1 < I_2 = I_4 = I_5 < I_3$	$I_1 < I_2 = I_4 = I_5 < I_3$
Q3	$I_3 < I_1 < I_2 < I_4 = I_5$	$I_3 < I_1 < I_2 < I_4 = I_5$	$I_3 < I_1 < I_2 = I_4 = I_5$	$I_3 < I_1 < I_4 = I_5 < I_2$
Q4	$I_1 < I_3 < I_2 = I_4 = I_5$	$I_1 < I_3 < I_2 < I_4 = I_5$	$I_1 < I_3 < I_2 = I_4 = I_5$	$I_1 < I_3 < I_2 = I_4 = I_5$
Q5	$I_2 < I_4 = I_5 < I_1 < I_3$	$I_2 < I_4 = I_5 < I_1 < I_3$	$I_2 < I_4 = I_5 < I_1 < I_3$	$I_2 < I_4 = I_5 < I_1 < I_3$

Table 7-3: continued.

	$520 <= N_{avg} < 600$	$600 <= N_{avg} < 1100$	$1100 <= N_{avg} < 1300$	$1300 <= N_{avg}$
Q1	$I_1 < I_2 < I_4 = I_5 < I_3$	$I_1 < I_4 = I_5 < I_2 < I_3$	$I_1 = I_4 = I_5 < I_2 < I_3$	$I_4 = I_5 < I_1 < I_2 < I_3$
Q2	$I_1 < I_2 = I_4 = I_5 < I_3$	$I_1 < I_4 = I_5 < I_2 < I_3$	$I_1 < I_4 = I_5 < I_2 < I_3$	$I_1 < I_4 = I_5 < I_2 < I_3$
Q3	$I_3 < I_1 < I_4 = I_5 < I_2$	$I_3 < I_1 < I_4 = I_5 < I_2$	$I_3 < I_4 = I_5 < I_1 < I_2$	$I_3 < I_4 = I_5 < I_1 < I_2$
Q4	$I_1 < I_3 < I_2 = I_4 = I_5$	$I_1 < I_3 < I_4 = I_5 < I_2$	$I_1 < I_3 < I_4 = I_5 < I_2$	$I_1 < I_3 < I_4 = I_5 < I_2$
Q5	$I_1 < I_2 < I_4 = I_5 < I_3$	$I_1 < I_2 < I_4 = I_5 < I_3$	$I_1 < I_2 < I_4 = I_5 < I_3$	$I_1 < I_2 < I_4 = I_5 < I_3$

Table 7-4: The relative performances when Pg is varied.

	$Pg < 2K$	$2K <= Pg < 4K$	$4K <= Pg < 10K$
Q1	$I_1 < I_4 = I_5 < I_2 < I_3$	$I_1 < I_2 < I_4 = I_5 < I_3$	$I_1 < I_2 = I_4 = I_5 < I_3$
Q2	$I_1 < I_4 = I_5 < I_2 < I_3$	$I_1 < I_2 < I_4 = I_5 < I_3$	$I_1 = I_2 = I_4 = I_5 < I_3$
Q3	$I_3 < I_1 < I_4 = I_5 < I_2$	$I_3 < I_1 < I_2 < I_4 = I_5$	$I_3 < I_1 < I_2 = I_4 = I_5$
Q4	$I_1 < I_3 < I_2 < I_4 = I_5$	$I_1 < I_3 < I_2 < I_4 = I_5$	$I_1 < I_3 < I_2 = I_4 = I_5$
Q5	$I_1 < I_2 = I_4 = I_5 < I_3$	$I_2 < I_4 = I_5 < I_1 < I_3$	$I_2 < I_4 = I_5 < I_3 < I_1$

Table 7-5: The relative performances when n is varied.

	$n < 7$	$7 <= n < 10$	$10 <= n$
Q2	$I_1 < I_2 < I_4 = I_5 < I_3$	$I_2 < I_1 = I_4 = I_5 < I_3$	$I_2 < I_4 = I_5 < I_1 < I_3$

CHAPTER 8
SUMMARY

In this dissertation, we have presented our approaches to modeling time concepts and constraints found in advanced database applications using an OO semantic model. Our research effort has concentrated on four areas: temporal knowledge model, temporal query language design, temporal algebra, and implementation techniques. With respect to the model, OSAM*/T, we use object instance time-stamping technique and the notions of Start-time and End-time to record the evolution of object instances. We also use temporal knowledge rules to capture complex temporal requirements and constraints. Temporal knowledge rules are also modeled as objects and can be updated as their applicabilities change in time. Both temporal knowledge rules and their historical versions can be automatically triggered by various knowledge base operations.

For querying an OO temporal database, we introduce OQL/T as the high-level query language. Formulating temporal queries in OQL/T can be specified simply by patterns of object associations in some time domains. Several temporal functions and interval comparison operators are introduced in the language so that the user can specify temporal requirements more easily.

To provide a solid mathematical foundation for temporal database processing, we introduce the TA-algebra as a formal language. The language is featured by its closure property and pattern-based specification and processing. TA-algebra operators perform operations based on snapshot semantics; high-level temporal queries as well as time

specifications using interval comparison operators [ALL84] can all be decomposed into algebraic representations for a uniform treatment by the operators.

Lastly, we propose a Delta-Instance, Multi-Snapshot storage model for saving storage space and achieving processing efficiency. In this model, temporal data are compressed as delta instances before they are stored; they are also partitioned into blocks based on time intervals so that any search for temporal data of a specified time interval can be carried out efficiently. Lengthy traversal of historical chain for materializing temporal data is avoided by storing a snapshot of the most recent data for each TDB. Furthermore, the amount of temporal data in each TDB is fixed, thus simplifying the problem of indexing a dynamically growing temporal database. Additionally, since temporal data of each TDB is self-contained, there is a greater potential for achieving parallel processing of temporal data using the DIMS model.

In order to understand the relative merits of our proposed approach and some existing techniques used for storing and processing temporal data, we evaluated their storage requirements and performances in processing a number of benchmark queries. The results show that our approach consumes the least amount of storage space and correspondingly has the best performance in materializing temporal data of a time interval. In processing temporal queries, our approach also has the best performance in several query types and are comparable to the other techniques in the other cases.

REFERENCE LIST

[ABR74] J.R. Abrial, *Data Semantics, Data Base Management*, North-Holland, Amsterdam, 1974, pp. 1-59.

[AHN86] I. Ahn, "Towards an Implementation of Database Management Systems with Temporal Support," Proc. of the IEEE Data Eng. Int'l Conf., 1986, pp. 374-381, Los Angeles, California.

[AHN88] I. Ahn, and R. Snodgrass, "Partitioned Storage for Temporal Databases," Information Systems Vol. 13 No. 4, 1988, pp. 369-391.

[ALA89a] A.M. Alashqur, S.Y.W. Su, and H. Lam, "OQL--An Object-Oriented Query Language," Proc. of the VLDB Int'l Conf., 1989, pp. 433-442, Amsterdam, the Netherland.

[ALA89b] A. M. Alashqur, "A Query Model and Query and Knowledge Definition Languages for Object-Oriented Databases," Ph.D. dissertation, Department of Electrical Engineering, University of Florida, Gainesville, Florida, 1989.

[ALL84] J.F. Allen, "Towards a General Theory of Action and Time," Journal of Artificial Intelligence Vol. 23, 1984, pp. 123-154.

[BAT84] Batory, D.S., and Alejandro P. Buchmann, "Molecular Objects, Abstract Data Types, and Data Models: A Framework," Proc. of the VLDB Int'l Conf., 1984, pp. 172-184, Singapore.

[BJO89] A. Bjornerstedt, and C. Hutten, "Version Control in an Object Oriented Architecture," Chapter 18 in *Object-Oriented Concepts, Databases, and Applications* ed. W. Kim and F.H. Lochovsky, 1989, New York, ACM Press.

[BLA77] M.W. Blasgen, and K.P. Eswaran, "Storage and Access in Relational Data Bases," IBM System Journal, Vol. 16, No. 4, 1977, pp. 176-188.

[BOL82] A. Bolour, T.L. Anderson, L.J. Deketser, and H.K.T. Wong, "The Role of Time in Information Processing: A Survey," ACM SIGMOD Rec. Vol. 11, No. 1, 1982, pp. 28-48.

[BOO86] G. Booch, "Object-Oriented Development," IEEE TSE, Vol. SE-12, No. 2, 1986.

[CHA90] U.S. Chakravarthy, and S. Nesson, "Making An Object-Oriented DBMS Active: Design, Implementation, and Evaluation of a Prototype," in Proc. of the Int'l Conf. on Extended Data Base Technology, 1990, Venice Italy.

[CHE76] P.P.S. Chen, "The Entity Relationship Model: Towards a Unified View of Data," ACM TODS, Vol. 1, No. 1, 1976.

[CHE93a] H.M Chen, and S.Y.W. Su, "Temporal Association Algebra: A Mathematical Foundation for Object-Oriented Temporal Data/Knowledge Bases" in preparation.

[CHE93b] H.M Chen, and S.Y.W. Su, "A Delta-Instance, Multi-Snapshot Storage Model for Object-Oriented Temporal Databases" in preparation.

[CHU90] H.S. Chuang, "Operational Rule Processing in a Prototype OSAM* Knowledge Base Management System," Master thesis, Department of Computer and Information Sciences, University of Florida, Gainesville, Florida, 1990.

[CLI87] J. Clifford, and A. Croker, "The Historical Relational Data Model (HRDM) and Algebra Based on Lifespan," IEEE Proc. of the Data Eng Int'l Conf., 1987, pp. 528-537, Los Angeles, California.

[CLI85] J. Clifford, and A.U. Tansel, "On an Algebra for Historical Relational Databases: Two Views," Proc. of the ACM SIGMOD Int'l Conf., 1985, pp. 247-265, Portland, Oregon.

[CLI83] J. Clifford, and D.S. Warren, "Formal Semantics for Time in Databases," ACM TODS, Vol. 8, No. 2, 1983, pp. 214-254.

[COP84] G. Copeland, and D. Maier, "Making Smalltalk a Database System," Proc. of the ACM SIGMOD Int'l Conf., 1984, pp.316-325, Boston, Massachusetts.

[DAT88] C.J. Date, "A Proposal for Adding Date and Time Support to SQL," SIGMOD Rec., Vol.17, No.2, June 1988, pp.53-76.

[DAY88a] U. Dayal, "Active Database Management Systems," Proc. of the 3rd Int'l Conf. on Data and Knowledge Bases, 1988, pp. 150-169.

[DAY88b] U. Dayal, A. Buchmann, and D. McCarthy, "Rules are Objects Too: A Knowledge Model for an Active, Object-Oriented Database Management System," Proc. of the Int'l Workshop on Object-Oriented Database Systems, 1988.

[DE87] S. De, S. Pan, and A. Whinston, "Temporal Semantics and Natural Language Processing in a Decision Support System," Information Systems, Vol. 12, No. 1, 1987, pp. 29-47.

[DEA86] T.L. Dean, and D.V. McDermott, "Temporal Data Base Management," Artificial Intelligence, Vol. 25, No. 1, 1986, pp. 1-55.

[DIT86] K.R. Dittrich, A.M. Kotz, and J.A. Mulle, "An Event/Trigger Mechanism to Enforce Complex Consistency Constraints in Design Databases," ACM SIGMOD Rec. Vol. 15, No. 3, 1986, pp. 22-36, Washington DC.

[EAS86] M.C. Easton, "Key-Sequence Data Sets on Indelible Storage," IBM Journal, Vol. 30, No. 3, 1986, pp. 230-241.

[ELM90a] R. Elmasri, I. El-Assal, and V. Kouramajian, "Semantics of Temporal Data in an Extended ER Model," in ER Conf., 1990, pp. 234-242.

[ELM91] R. Elmasri, Y.J. Kim, and G.T. Wuu, "Efficient Implementation Techniques for the Time Index," Proc. of the IEEE Data Eng. Int'l Conf., 1991, pp. 102-111, Kobe, Japan.

[ELM92] R. Elmasri, M. Jaseemuddin, and V. Kouramajian, "Partitioning of Time Index for Optical Disks," Proc. of IEEE Data Eng. Int'l Conf., 1992, pp. 574-583, Tempe, Arizona.

[ELM90b] R. Elmasri, and C. Udomwongsa, "An ER Interface for an Object-Oriented System," Tech. Report #UH-CS-90-30, University of Houston, Texas, 1990.

[ELM90c] R. Elmasri, and C. Udomwongsa, "An ER Interface for an Object-Oriented System," Tech. Report #UH-CS-90-30, University of Houston, Texas, 1990.

[ELM90d] R. Elmasri, and G.T.J. Wuu, "The Time Index: An Access Structure for Temporal Data," Proc. of the VLDB Int'l Conf., 1990, Brisbane, Australia.

[GAD86] S.K. Gadia, "Toward a Multihomogeneous Model for a Temporal Database," Proc. of the IEEE Data Eng. Int'l. Conf., 1986, pp. 390-397, Los Angeles, California.

[GAD88] S.K. Gadia, "A Homogeneous Relational Model and Query Languages for Temporal Databases," ACM TODS, Vol. 13, No. 4, 1988, pp. 418-448.

[GUN90] H. Gunadhi, and A. Segev, "Efficient Indexing Methods for Temporal Relations," Technical Report LBL-28798, University of California, Berkeley, California, 1990.

[GUO91] M. Guo, S.Y.W. Su, and H. Lam, "An Association Algebra for Processing Object-Oriented Databases," Proc. of the IEEE Data Eng. Int'l. Conf., 1991, pp.344-354, Kope, Japan.

[HAM78] M. Hammer, and D. McLeod, "The Semantic Data Model: A Modeling Mechanism for Database Applications," Proc. of the ACM SIGMOD Conf., 1978, pp. 144-156, Austin, Texas.

[HOA85] A.S. Hoagland, "Information Storage Technology a Look at the Future," IEEE Computer, Vol.18, No.7, 1985, pp.60-67.

[JEN92a] C.S. Jensen, J. Clifford, S.K. Gadia, A. Segev, and R.T. Snodgrass, "A Glossary of Temporal Database Concepts," SIGMOD Rec., Vol. 21, No. 3, 1992, pp. 35-43.

[JEN92b] C.S. Jensen, and R.T. Snodgrass, "Proposal for a Data Model for the Temporal Structured Query Language," TEMPIS Technical Report No. TR 92-37, Department of Computer Science, University of Arizona, Tucson, Arizona, 1992.

[JEN92c] C.S. Jensen, M.D. Soo, and R.T. Snodgrass, "Unification of Temporal Data Models," TR 92-15, Technical Report, Department of Computer Science, University of Arizona, Tucson, Arizona, 1992.

[JEN92d] C.S. Jensen, M.D. Soo, and R.T. Snodgrass, "Extending Normal Forms to Temporal Relations," TR 92-17, Technical Report, Department of Computer Science, University of Arizona, Tucson, Arizona, 1992.

[KHO86] S. Khoshafian, and G.P. Copeland, "Object Identity," Proc. of the OOPSLA Conf., 1986, pp. 30-38, Orlando, Florida.

[KHO90] S. Khoshafian, and R. Abnous, *Object Orientation Concepts, Languages, Databases, User Interfaces*, John Wiley & Sons Inc., 1990.

[KIM89] W. Kim, and F.H. Lochovsky, *Object-Oriented Concepts, Databases, and Applications*, New York, ACM Press, 1989.

[KIM90a] J. Kim, H. Yoo, and Y. Lee, "Design and Implementation of A Temporal Query Language with Abstract Time," Information Systems, Vol.15, No. 3, 1990, pp.349-357.

[KIM90b] W. Kim, "Object-Oriented Databases: Definition and Research Directions," IEEE TKDE, Vol.2, No. 3, 1990, pp. 327-341.

[KOL89] C. Kolovson, and M. Stonebraker, "Indexing Techniques for Historical Databases," Proc. of the IEEE Data Eng. Int'l Conf., 1989, pp. 127-137.

[LEV87] H.J. Levesque, "A View of Knowledge Representation," in *On Knowledge Base Management Systems*, M.L. Brodie and J. Mylopoulos (eds), New York, Springer-Verlag, 1987.

[LOM89] D. Lomet, and B. Salzberg, "Access Methods for Multiversion Data," Proc. of the ACM SIGMOD Int'l Conf., 1989, pp. 315-324, Portland, Oregon.

[LOR88] N. A. Lorenitzos, and Johnson R. G., "Extending Relational Algebra to Manipulate Temporal Data." Information Systems, Vol. 13, No. 3, 1988, pp. 289-296.

[LUM84] V. Lum, P. Dadam, R. Erbe, J. Guenauer, and P. Pistor, "Designing DBMS Support for The Temporal Dimension," Proc. of the ACM SIGMOD Int'l Conf., 1984, pp.115-130, Boston, Massachusetts.

[MAI86] D. Maier, and J. Stein, "Development of an Object Oriented DBMS," Proc. of the OOPSLA Conf., 1986, pp. 472-482, Orlando, Florida.

[MAI91] R. Maiocchi, and B. Pernici, "Temporal Data Management Systems: A Comparative View," IEEE TKDE, Vol. 3, No. 4, 1991, pp. 504-524.

[McK86] E. McKenzie, "Bibliography: Temporal Databases," ACM SIGMOD Rec., Vol. 15, No. 4, 1986, pp. 40-52.

[McK90] E. Mckenzie, and R. Snodgrass, "Schema Evolution and the Relational Algebra," Information Systems, Vol. 15, No. 2, 1990, pp. 207-232.

[McK91] L.E. McKenzie, and R.T. Snodgrass, "Evaluation of Relational Algebras Incorporating the Time Dimension in Databases," ACM Computing Surveys, Vol. 23, No. 4, 1991, pp. 501-543.

[NAV89] S. B. Navathe, and R. Ahmed, "A Temporal Relational Model and a Query Language," Information Sciences, Vol. 48, No. 2, 1989, pp. 57-73.

[NIE89] O. Nierstrasz, "A Survey of Object-Oriented Concepts," in *Object-Oriented Concepts, Databases, and Applications*, ed. W. Kim and F.H. Lochovsky, New York, ACM Press, 1989.

[ROC75] M. J. Rochkind, "The Source Code Control System," IEEE TSE, Vol.SE-1, No.4, 1975, pp. 364-370.

[ROS91] E. Rose, and A. Segev, "TOODM--a Temporal Object-Oriented Data Model with Temporal Constraints," Proc. of the Int'l Conf. on the ER approach, 1991, pp. 205-229.

[ROS92] E. Rose, and A. Segev, "TO-Algebra--a Temporal Object-Oriented Algebra," Technical Report LBL-32013, University of California, Walter A. Haas School of Business, Berkeley, California, 1992.

[ROT87] D. Rotem, and A. Segev, "Physical Organization of Temporal Data," Proc. of the IEEE Data Eng. Int'l Conf., 1987, pp. 547-553, Los Angeles, California.

[SAR86] S.K. Sari, C.W. , Kaufman, and J.E. Somers, "Using History Information to process Delayed Database Updates," Proc. of the VLDB Int'l Conf., 1986, pp. 71-78, Kyoto, Japan.

[SAR90] N.L. Sarda, "Algebra and Query Language for A Historical Data Model," the Computer Journal, Vol. 33, No. 1, 1990, pp. 11-18.

[SEG87] A. Segev, and A. Shoshani, "Logical Modeling of Temporal Data," Proc. of the ACM SIGMOD Int'l Conf., 1987, pp. 454-466, San Francisco, California.

[SEG88] A. Segev, and A. Shoshani, "Functionality of Temporal Data Models and Physical Design Implications," Proc. of the IEEE Data Eng. Int'l Conf., Special Issue on Temporal Databases, Vol. 11 No. 4, 1988, pp. 38-45.

[SEG88a] A. Segev, and A. Shoshani, "The Representation of a Temporal Data Model in the Relational Environment," in *Lecture Notes in Computer Science*, Vol.339, M. Rafanelli, J.C. Klensin and P. Svensso, Eds. New York: Springer-Verlag, 1988, pp.39-61.

[SEG 89] A. Segev, and H. Gunadhi, "Event-Join Optimization in Temporal Relational Databases," Proc. of the VLDB Int'l Conf., 1989, pp. 205-215, Amsterdam, the Netherland.

[SEV76] D.G. Severance, and G. M. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases," ACM TODS, Vol.1, No.3, 1976, pp. 256-267.

[SIL90] A. Silberschatz, M. Stonebraker, and J.D. Ullman, "Database Systems: Achievements and Opportunities," ACM SIGMOD Rec., Vol. 19, No. 4, 1990, pp. 6-22.

[SIN90] M. Singh, "Transaction Oriented Rule Processing in An Object-Oriented Knowledge Base Management System," Master Thesis, Department of Electrical Engineering, University of Florida, Gainesville, Florida, 1990.

[SMI77] J.M. Smith, and D.C.P. Smith, "Database Abstractions: Aggregation and Generalization," ACM TODS, Vol. 2, No.2, 1977, pp.105-133.

[SNO85] R. Snodgrass, and I. Ahn, "A Taxonomy of Time in Database" Proc. of the ACM SIGMOD Int'l Conf., 1985, pp. 236-246, Portland, Oregon.

[SNO86] R. Snodgrass, "Research Concerning Time in Databases: Project Summaries," ACM SIGMOD Rec., Vol. 15, No. 4, 1986, pp. 19-39.

[SNO87] R. Snodgrass, "The Temporal Query Language TQuel," ACM TODS, Vol. 12, No. 2, 1987, pp. 247-297.

[SNO90] R. Snodgrass, "Temporal Databases: Status and Research Directions," ACM SIGMOD Rec., Vol. 19, No. 4, 1990, pp. 83-89.

[SOO91] M.D. Soo, "Bibliography on Temporal Databases," ACM SIGMOD Rec., Vol. 20, No. 1, 1991, pp. 14-23.

[STO87] M. Stonebraker, "The Design of the Postgres Storage System," Proc. of the VLDB Int'l Conf., 1987, pp. 289-300, Brighton, England.

[SU89] S.Y.W. Su, H. Lam, V. Krishnamurthy, "An Object-Oriented Semantic Association Model (OSAM*)" Chapter 17 in Artificial Intelligence: Manufacturing Theory and Practice, edited by S.T. Kumara, A.L. Soyster, and R.L. Kashyap, Published by the Institute of Industrial Engineers, Industrial Engineering and Management Press, Norcross, GA, 1989.

[SU91] S.Y.W. Su, and H.M. Chen "A Temporal Knowledge Representation Model OSAM*/T and its Query Language OQL/T," Proc. of the VLDB Int'l Conf., 1991, Barcilona, Spain.

[SU92] S.Y.W. Su, and H.M. Chen, "Temporal Rule Specification and Management in Object-oriented Knowledge Bases," Technical Report TR-92-026, CIS Dept., University of Florida, Gainesville, Florida, 1992.

[SU93] S.Y.W. Su, H. Lam, and M. Guo, "Association Algebra: A Mathematical Foundation for Temporal Object-Oriented Knowledge Bases," in press.

[TAN86] A.U. Tansel, "Adding Time Dimension to Relational Model and Extending Relational Algebra" Information Systems, Vol. 11, No.4, 1986 pp.343-355.

[TUZ90] A. Tuzhilin, and J. Clifford, "A Temporal Relational Algebra as a Basis for Temporal Relational Completeness," Proc. of the VLDB Int'l Conf. , 1990, pp. 13-23, Brisbane, Australia.

[VOS91] G. Vossen, "Bibliography on Object-Oriented Database Management," ACM SIGMOD Rec., Vol. 20, No. 1, 1991, pp 24-46.

[WUU92] G.T.J. Wuu, and U. Dayal, "A Uniform Model for Temporal Object-Oriented Databases," Proc. of the IEEE Data Eng Int'l Conf., 1992, pp. 584-593, Tempe, Arizona.

[YAS91] R. Yaseen, S.Y.W. Su, and H. Lam, "An Extensible Kernel Object Management System," Proc. of the OOPSLA Conf., 1991.

[ZDO86] S. Zdonik, and P. Wegner, "Language and Methodology for Object-Oriented Database Environments," Proc. 19th Annual Hawaiian Conference on Systems Science, 1986.

[ZDO90] S. Zdonik, and D. Maier, "Fundamentals of Object-Oriented Databases" in *Readings in Object-Oriented Database Systems*, ed. S. Zdonik and D. Maier, Los Altos, California, Morgan Kaufman Publisher, 1990.

APPENDIX A
BNF OF THE OQL/T

```
okmql_statement : block '?';
block : [BEGINS TRANSACTION]
            statement { statement }
        [ENDS TRANSACTION];

/* Basic query block structure of OQL/T. */
statement : temporal_context_specification
            operation_subclause;

temporal_context_specification      : when_clause
                                      context_specification
                                    | when_clause
                                      context_specification
                                      inter_cross_reference;

context_specification:
        CONTEXT association_pattern {',' association_pattern }
                where_clause;

where_clause :  | WHERE search_condition;

/* operation clause. */
operation_subclause   : system_defined_op_clauses  |  user_defined_op_clauses;

/* association pattern */
association_pattern    : link_expr
                       | association_pattern lbranch '(' pattern_list ')'
                       | association_pattern lbranch '(' pattern_list ')' rbranch link_expr
                       | '(' pattern_list ')' rbranch link_expr;

lbranch: assoc_op AND  |  assoc_op OR  |  non_assoc_op AND  |  non_assoc_op OR;

rbranch : AND assoc_op  |  AND non_assoc_op  |  OR assoc_op  |  OR non_assoc_op;

pattern_list : link_expr  |  pattern_list ',' association_link;

link_expr : link_term { non_assoc_op link_term };

link_term : lfactor { assoc_op lfactor };

lfactor : IDENTIFIER  |  IDENTIFIER '[' search condition ']'  |  '(' link_expr ')';
```

```
assoc_op : '*';

non_assoc_op : '!';

/* Syntax for search_condition. */
search_condition : boolean_term { OR boolean_term };

boolean_term : boolean_factor { AND boolean_factor};

boolean_factor : boolean_primary  |  NOT boolean_primary;

boolean_primary : predicate  |  '(' search_condition ')';

predicate : comparison_argument comp_op value_specification;

comp_op : '>'  |  '<'  |  '='  |  '!='  |  '>='  |  '<=';

comparison_argument : value_expression;

value_expression : term  |  value_expression '+' term  |  value_expression '-' term;

term : factor  |  term '*' factor  |  term '/' factor;

factor : primary  |  '+' primary  |  '-' primary;

primary : attribute_specification  |  value_specification  |  '(' value_expression ')';

value_specification : CHAR_VALUE  |  NUMERIC_VALUE  |  NULL;

attribute_specification : IDENTIFIER '[' IDENTIFIER ']';

/* operation clause */
system_defined_op_clauses : retrieve_clause  |  modification_block;

modification_block    :[BEGINS TRANSACTION]
                       modification_clause {modification_clause}
                       [ENDS TRANSACTION];

modification_clause   : insert_clause|update_clause|delete_clause|associate_clause
                      | disassociate_clause|instantiate clause|Del_clause|Cor_clause;

retrieve_clause : RETRIEVE retrieve_list;

retrieve_list : '*' | ALL | select_spec {',' select_spec};

select_spec : value_expression | renamed_form;

renamed_form : IDENTIFIER ':' value_expression;

insert_clause : INSERT '(' attribute_value { ',' attribute_value} ')';
```

```
update_clause : UPDATE '(' attribute_value { ',' attribute_value} ')';

delete_clause : DELETE '(' IDENTIFIER { ',' IDENTIFIER } ')';

associate_clause : ASSOCIATE'('IDENTIFIER','IDENTIFIER')' ['['IDENTIFIER']'];

disassociate_clause :
        DISASSOCIATE '(' IDENTIFIER ',' IDENTIFIER ')' [ '[' IDENTIFIER ']' ];

instantiate_clause : INSTANTIATE '(' CONTEXT lfactor {',' attribute_value} ')';

Cor_clause :
        COROBJ '(' '('IDENTIFIER { ',' IDENTIFIER}')' ','
                '('value_expression {',' value_expression}')' ')';

Del_clause : DELOBJ '(' attribute_value { ',' attribute_value } ')';

attribute_value : IDENTIFIER '=' value_expression;

user_defined_op_clause : IDENTIFIER { '(' IDENTIFIER { ',' IDENTIFIER } ')' };

/* temporal conditions */
when_clause :  | when_clause_1;

when_clause_1 : WHEN references  |  AT time_point_reference;

references : time_interval_reference  |  data_reference;

time_interval_reference: time_int;

time_int : 'T' '[' scalar_exp ',' scalar_exp ']';

scalar_exp : specified_time  |  time_exp;

specified_time : time_point {'-' specified_time};

time_point : scalar;

scalar : NUMERIC_VALUE;

time_exp : TIME '(' time_para ')';

time_para : NOW scalar_type;

scalar_type : '+' scalar  |  '-' scalar;

time_point_reference: scalar_exp;

data_reference : temp_func | temp_func  WHERE boolean_condition | mov_win;
```

```
boolean_condition     : temp_func int_op temp_func | temp_func int_op time_int
                      | time_int int_op temp_func;

temp_func : INTERVAL '(' temp_func1 ')';

temp_func1 : association_pattern  | temporal_functions '(' association_pattern ')';

temporal_functions    : FORMER | NEXT | START | END | FIRST | NTH | LAST
                      | B-FIRST | B-NTH | B-LAST;

int_op : BEFORE|AFTER|PRECEDE|FOLLOW|P-CROSS|F-CROSS|CROSS
       | EQUAL|CONTAIN|WITHIN|O-CONTAIN|I-WITHIN|L-CONTAIN
       | L-WITHIN|R-CONTAIN|R-WITHIN;

mov_win       : mov_func | mov_func mov_range | mov_func advance_op
              | mov_func mov_range advance_op;

mov_func : INTERVAL '(' parameter ')' move_op;

mov_op : ANY scalar time_unit | EVERY scalar time_unit;

mov_range: WITHIN time_int;

advance_op : INCREMENT_BY scalar time_unit;

time_unit : SEC | MIN | HOUR | DAY | WEEK | MONTH | YEAR;

inter_cross_reference :       set_operators target_class
                              temporal_context_specification;

set_operators : INTERSECT | NT-INTERSECT  |  DIFFERENCE  | UNION;

target_class : '(' class_specification ')';

class_specification : IDENTIFIER {',' IDENTIFIER};
```

APPENDIX B
BNF OF THE KNOWLEDGE DEFINITION LANGUAGE FOR OSAM*/T

```
schema_declaration : SCHEMA schema_name
                     domain_declaration
                     entity_declaration
                   END schema_name ;

schema_name : IDENTIFIER;

domain_declaration : DOMAIN_CLASSES
                          classname ':' data_type
                          { ';' classname ':' data_type }
                     END DOMAIN_CLASSES;

classname : IDENTIFIER;

data_type : INTEGER|REAL|STRING|CHARACTER|BOOLEAN|COMPUTE;

entity_declaration      : ENTITY_CLASSES
                          entity_class_block
                          { ';' entity_class_block }
                        END ENTITY_CLASSES;

entity_class_block : ENTITY_CLASS classname
                          association_declaration
                          operation_declaration
                          temporal_rule_declaration
                     END classname ;

association_declaration : ASSOCIATION_SECTION
                               { association_specification ';' }
                          END ASSOCIATION_DECLARATION ;

association_specification      : generalization_declaration
                               | aggregation_declaration
                               | interaction_declaration
                               | crossproduct_declaration
                               | composition_declaration ;

generalization_declaration :
      GENERALIZATION OF '{' classname {';' classname } '}' g_constraints;

g_constraints : classname ',' classname ':' g_const {';'classname','classname':'g_const};
```

```
g_const : SX | SI | SE | ST-SS;

aggregation_declaration :
        AGGREGATION OF '{' attribute_declaration { ';' attribute_declaration } '}';

attribute_declaration : attribute_name ':' classname '(' A_const ')';

attribute_name : IDENTIFIER;

A_const : OPTIONAL | TOTAL;

interaction_declaration :
        INTERACTION OF '{' classname ';' classname { ';' classname } '}' i_constraints;

i_constraints : classname ',' classname ':' cardinality
                    { ';' classname ',' classname ':' cardinality };

cardinality : CHAR_VALUE;

crossproduct_declaration : CROSSPRODUCT OF '{' classname { ';' classname } '}';

composition_declaration : COMPOSITION OF '{' classname {';' classname } '}';

operation_declaration :
        OPERATION_SECTION
                func_oper { ';' func_oper }
        END OPERATION_SECTION;

func_oper : ufunction | uoperation;

ufunction : function_name '(' arguments ')' ':' data_type;

function_name : IDENTIFIER;

arguments : variable_name ':' classname { ';' variable_name ':' classname };

variable_name : IDENTIFIER;

uoperation : operation_name '(' arguments ')';

operation_name : dml_clause | message_spec;

dml_clause : delete_clause | update_clause | insert_clause | retrieve_clause
            | operation4 | operation5;

delete_clause : DELETE_OBJECT classNlist | DELETE_INSTANCE classNlist;

classNlist : '(' class_names ')' | classname;

class_names : class_names ',' classname | classname;
```

```
update_clause :  UPDATE '(' classname ',' attribute_value_list ')';

attribute_value_list : attribute_value_list ',' attribute_value | attribute_value;

attribute_value: attribute_name '=' value_expression
                | attribute_name comp_op value_expression;

value_expression : value_expression '+' term | value_expression '-' term | term;

term : term '*' factor | term '/' factor | factor;

factor : '+' primary | '-' primary | primary;

primary : attribute_specification | value_specification
        | '(' value_expression ')' | IDENTIFIER '[' search_condition ']';

attribute_specification : attribute_specification '.' IDENTIFIER | IDENTIFIER;

value_specification : CHAR_VALUE | NUMERIC_VALUE | NULL;

search_condition : search_condition ',' boolean_term | boolean_term;

boolean_term : boolean_term AND boolean_factor | boolean_factor;

boolean_factor : NOT boolean_primary | boolean_primary;

boolean_primary : predicate;

predicate : comparison_predicate;

comparison_predicate : comparison_argument comp_op value_expression;

comparison_argument : value_expression;

comp_op : '>' | '<' | '=' | '!=' | '>=' | '<=';

insert_clause : INSERT_INSTANCE '(' target_and_source ')';

target_and_source : target ',' source;

target : classname;

source : classname;

retrieve_clause : RETRIEVE IDENTIFIER;

message_spec : MESSAGE  messagebody;

messagebody : '(' CHAR_VALUE ')';
```

```
temporal_rule_declaration :
        TEMPORAL_RULE_SECTION
                domain_rule { ';' domain_rule }
        END TEMPORAL_RULE_SECTION ;

/* rule section */
domain_rule   : RULE rule_id
                valid_time
                'triggered' '(' trig_conds ')'
                rule_body
                END;

rule_id : NUMERIC_VALUE;

valid_time: 'Valid_T' '['start_time ',' end_time']'

start_time : scalar;

scalar : NUMERIC_VALUE;

end_time : '-' | scalar;

trig_conds : trig_conds ',' trig_cond | trig_cond;

trig_cond :       option operation;

option : 'before' | 'after' | 'parallel' | 'immediate_after';

operation : operation1 | operation2 | operation3 | operation4 | operation5;

operation1 : UPDATE '(' argument2 ')';

operation2 : ins_arg '(' argument1 ')';

ins_arg : INSERT_OBJECT | INSERT_INSTANCE;

operation3 : del_arg '(' argument1 ')';

del_arg : DELETE_OBJECT | DELETE_INSTANCE;

operation4 : RETRIEVE '(' argument2 ')';

operation5 : USER_DEFINED '(' CHAR_VALUE ')';

argument1 : argument1 ',' class_reference | class_reference;

argument2 : argument2 ',' attribute | attribute;

attribute : class_reference '.' attribute_reference | attribute_reference;

attribute_reference : IDENTIFIER;
```

```
class_reference : IDENTIFIER;

/* rule body specification */
rule_body : statement;

statement : not_exist_expression | k_rule;

temp_subdb    : when_clause
                subdb_definition;

subdb_definition : link_pattern | CONTEXT link_pattern;

link_pattern : link_expr | link_pattern lbranch '(' pattern_list ')'
                | link_pattern lbranch '(' pattern_list ')' rbranch link_expr
                | '(' pattern_list ')' rbranch link_expr;

link_expr : link_expr non_assoc_op link_term | link_term;

lbranch : *AND | *OR | !AND | !OR;

rbranch : AND* | AND! | OR* | OR!;

link_term :  link_term assoc_op lfactor | lfactor;

lfactor : IDENTIFIER | IDENTIFIER '[' search_condition ']' | '(' link_expr ')';

pattern_list : pattern_list ',' link_pattern | link_pattern;

assoc_op : '*';

non_assoc_op : '!';

cardinality_const : MAPPING classNlist ':' classNlist cardinality;

/* temporal conditions */
when_clause :  | when_clause_1;

when_clause_1 : WHEN references | AT time_point_reference;

references : time_interval_reference | data_reference;

time_interval_reference: time_int;

time_int : T '[' scalar_exp ',' scalar_exp ']';

scalar_exp : specified_time | time_exp;

specified_time : scalar;

time_exp : TIME '(' time_para ')';
```

```
time_para : NOW scalar_type;

scalar_type : '+' scalar | '-' scalar;

time_point_reference: scalar_exp;

data_reference: temp_func | temp_func  WHERE boolean_condition | mov_win;

boolean_condition     : temp_func int_op temp_func | temp_func int_op time_int
                      | time_int int_op temp_func;

temp_func : INTERVAL '(' temp_func1 ')';

temp_func1 : link_pattern | temporal_functions '(' link_pattern ')';

temporal_functions    : FORMER | NEXT | START | END | FIRST | NTH | LAST
                      | B_FIRST | B_NTH | B_LAST;

int_op : BEFORE|AFTER|PRECEDE|FOLLOW|P-CROSS|F-CROSS|CROSS
       | EQUAL|CONTAIN|WITHIN|O-CONTAIN|I-WITHIN|L-CONTAIN
       | L-WITHIN|R-CONTAIN|R-WITHIN;

mov_win         : mov_func | mov_func mov_range | mov_func advance_op
                | mov_func mov_range advance_op;

mov_func : INTERVAL '(' predicate ')' move_op;

mov_op : ANY scalar time_unit | EVERY scalar time_unit;

mov_range: WITHIN time_int;

advance_op : INCREMENT_BY scalar time_unit;

time_unit : SEC | MIN | HOUR | DAY | WEEK | MONTH | YEAR;

not_exist_expression : NOT_EXIST temp_subdb;

k_rule : 'condition' guard_condition
         'action' action_part
       | 'condition' guard_condition
         'otherwise' otherwise_part
       | 'condition' guard_condition
         'action' action_part
         'otherwise' otherwise_part ;

guard_condition : boolean_exp | guarded_boolean_exp;

guarded_boolean_exp : guard_exp '|' guarded_exp;

guard_exp : boolean_exp {',' boolean_exp};
```

guarded_exp : boolean_exp

boolean_exp : not_exist_expression | temp_subdb | quantifier_exp;

quantifier_exp : existential_exp | universal_exp;

existential_exp : 'exisit' var_list 'in' temp_subdb ['suchthat'temp_subdb];

universal_exp : 'forall' var_list 'in' temp_subdb ['suchthat'temp_subdb];

action_part : action_part_of_state | action_part_of_transition;

action_part_of_state : not_exist_expression | temp_subdb | cardinality_const;

action_part_of_transition : dml_clause;

otherwise_part: not_exist_expression | temp_subdb | cardinality_const;

APPENDIX C
TDB INDEX

TDB Index (which is implemented by an AP-tree) for time intervals of order n will have the following features:

1. all internal nodes except the root have at most n children and at least 2 children,
2. the number of keys in each internal node is 1 less than that of its children,
3. the root can have at most n children, and at least 2 children, or none if it is a single node tree,
4. for a tree with d children, and a height of h, each of the first d-1 children is the root of a subtree where
   (1) all leaves of the d-1 subtrees are on the same level,
   (2) the d-1 subtrees have a height of h-1,
   (3) the internal nodes of the d-1 subtrees have n-1 keys

   for the rightmost subtree rooted at the dth child:
   (1) it has a height of at least one, and no more than h-1,
   (2) when its height reaches h-1 and each internal node has n-1 keys, the next key insertion into the TDB-tree will create a new right subtree.
5. access to the tree is either through the root or through the right-most leaf.

The algorithms for searching TDBs and inserting a TDB are given below. In the search algorithm we assume that the given time condition is legal.

Search Algorithm

```
Begin_search
    if no time condition is given
        goto RightMostLeaf;
        /* the query is evaluated against the current information which is pointed
        to by the pointer in the RightMostLeaf */
    else
        start from the Root and traverse down the tree to find the entry whose
        time interval matches with the given time condition;

    follow the pointers of the found TDB to retrieve the second level indices and
    perform the second level search;
End_search
```

InsertLeaf algorithm

```
Begin_insert
        if RootPtr = nil
                { create Root and insert Newkey; }
        else
                { retrieve RightMostLeaf; }

        if RightMostLeaf is not full
                { insert Newkey into RightMostLeaf; }
        else /* RightMostLeaf is full */
                {
                Create NewLeaf and insert Newkey;
                if the parent of the RightMostLeaf is not nil and at least one ancestor
                node of the RightMostLeaf is not full
                        {
                        find the lowest level non-full AncestorNode;
                        if the height between the AncestorNode and the RightMostLeaf is
                        the same as that of any subtree of this AncestorNode
                                {
                                insert the Newkey into the AncestorNode;
                                set the parent of the NewLeaf to this AncestorNode;
                                set the NewLeaf to be the RightMostLeaf;
                                }
                        else /* the height is different from any subtree of the ancestor
                        node*/
                                {
                                create InternalNode;
                                insert the Newkey to the InternalNode;
                                set the InternalNode to be the parent of the NewLeaf;
                                set the InternalNode to be the parent of the right most
                                child of the AncestorNode;
                                set the InternalNode to be the right most child of the
                                AncestorNode;
                                set the NewLeaf to be the RightMostLeaf;
                                }
                        }
                else
                /* the parent of the RightMostLeaf is nil; or */
                /* ancestor nodes of the RightMostLeaf is full */
                /* this also means that the right most subtree of the Root is full */
                        {
                        Create a NewRoot;
                        insert the NewKey to the NewRoot;
                        set the NewRoot to be the parent of the NewLeaf;
                        set the NewRoot to be the parent of the old Root;
                        set the NewLeaf to be the RightMostLeaf;
                        }
                }
End_insert;
```

APPENDIX D
ANALYTICAL FORMULAS FOR STORAGE CONSUMPTION

For the convenience of the readers, we list here all the variables and their explanations of section 7.4.

Notations:

N	The average number of attributes of each object class (or relation), among which, N-1 of them are non-synchronous and time-varying.
A	The average storage space in bytes for each attribute.
M	The average number of time-varying attributes involved in an evolution of each object instance. Each attribute has the same probability that it will be modified in each evolution of an object instance.
S	The total number of object instances (or surrogates) in an object class (or a relation).
N_{avg}	The average number of evolutions of an object instance (or surrogate) during an interval T.
T	A sample time period.
TB	The number of temporal data blocks during T, TB < N_{avg}.
C	The number of object classes (relations) in a database schema.
Pg	The size of a physical page.
n	The number of TDB's involved in a specified time interval.
S_{avg}	The average number of modified object instances at a time.

D.1 Object instance time-stamping

In object instance time-stamping, the storage requirement after N_{avg} evolution of object instances stored as temporal data blocks (TDB) can be derived as follows:

* storage space required for each object instance in a TDB,

$$S_o[1_instance_in_TDB] = (N+2)*A + (M+2)*A*Avg$$

where 2's accounts for the storage space of start-time and end-time stamps, (N+2)*A accounts for the storage space needed in the RDSA, (M+2)*A accounts for the storage space needed in the HDSA, M accounts for the average

number of changed attributes in one evolution, Avg is the average number of evolution in a TDB for each object instance and is equal to N_{avg}/TB.

* storage space required for S object instances of an object class in a TDB,
$S_o[S_instances_in_TDB] = S * A * [(N+2) + (M+2) * Avg]$

* storage space required for an object class within T duration:
$S_o[S_instances_in_T] = TB * S_o[S_instances_in_TDB]$
$= TB * S * A * [(N+2) + (M+2) * Avg]$

* storage space required for the temporal database within T duration:
$S_o[db_in_T] = C * S_o[S_instances_in_T]$
$= C * S * A * [TB * (N+2) + (M+2) * N_{avg}]$

D.2 Tuple time-stamping

In tuple time-stamping, the size of each temporal tuple is the same: (N+2)*A. Therefore, the storage requirement for a temporal database having C relations, S tuples in each relations and N_{avg} evolutions within T duration is $S_t[db_T]$, where

$S_t[db_T] = C * S * A * (N+2) * N_{avg}$

D.3 Attribute time-stamping

In attribute time-stamping, each (binary) relation contains only 4 attributes: the key which is a time-invariant attribute, one time-variant attribute, start-time and end-time stamps. Each original relation can be decomposed into N-1 binary relations if the N-1 attributes are not synchronous. That is, the data of a surrogate are distributed in the N - 1 binary relations. The probability that the tuples of a binary relation (or an attribute of the original relation) will be modified in each of the N_{avg} evolution of a surrogate is P (= M/(N-1)). This is because that each evolution will involve modifications of M attributes out of N-1 attributes. The storage requirement in this approach is $S_a[db_T]$, where

$S_a[db_T] = C * (N-1) * S * 4 * A * N_{avg} * P$

APPENDIX E
PARAMETERS VALUES USED IN THE ANALYSIS

Pg = 2K bytes, the average page size is varied between 1K to 10K;
M = 10, the average number of modified attributes of an object instance is varied between 1 to 41;
C = 50, the average number of classes in a database schema;
S = 10000, the average number of object instances is varied between 10K to 100K;
A = 4, the average size of an attribute in bytes;
TB = 15, the average number of temporal data blocks;
N = 20, the average number of attributes of an object instance is varied between 100 to 1000;
N_{avg} = 300, the average number of evolution of a surrogate;
S_{avg} = 1000, the average number of modified object instances at the same time is varied between 1 to 5001;
Avg = N_{avg}/TB;
L_{TDB} = ceil((N+2)*A/Pg) + ceil((2*M+2)*A*(N_{avg}/TB)/Pg);
L_T = ceil(N_{avg}/f);
L_{TB} = ceil(Avg/f);
H_s = ceil($\log_f$(S)), the height of the surrogate tree;
H_t = ceil($\log_f$(S*(N_{avg}+1)/S_{avg})), the height of the Time Index;
H_{ts} = ceil($\log_f$(N_{avg}+1)), the height of the local Time Index;
H_{ap} = ceil($\log_f$(N_{avg})), the height of the local AP-tree;
f = ceil(Pg/(A+4+4)); fan-out rate: "data + previous_pointer + next_pointer" or "data + left_node_pointer + right_node_pointer";
n = 1, the number of TDBs in a time interval is varied between 1 to 10;
NA = 100, the number of distinct attribute values is varied from 100 to 500;
k = S/NA, the number of qualified object instances is varied from 1 to S;

APPENDIX F
ANALYTICAL FORMULAS FOR MATERIALIZING TEMPORAL DATA

In this cost analysis, we focus only on the time needed (1) to read data from secondary storage to main memory and (2) to materialize temporal data for a time interval equivalent to a TDB duration and to the entire history.

F.1 Object instance time-stamping:

Materialize a TDB

* time required to materialize a TDB is the time needed to search for the data location, to read the data into main memory, and to materialize versions of each object instance in the TDB:

t[o_TDB] = 2 * t[access] + t[read_TDB] + t[mat_S_obj],

where 2 * t[access] accounts for the disk access time of the RDSA and HDSA of a TDB;

* time required to read a TDB:
t[read_TDB] = [ceil(S*A*(N+2)/Pg) + ceil(S*A*(M+2)*Avg/Pg)] * t[I/O];

* time required to materialize versions of an object instance in a TDB:
t[mat_a_obj] = (N+2) * t[s] + {(M+2) * (t[s]+t[rep]) + t[mov]} * Avg;

where (N+2)*t[s] accounts for the time to load the most recent version of an object instance from RDSA of a TDB, (M+2) accounts for the fact that there are an average of (M+2) attributes which need to be replaced to materialize each historical version of a temporal object instance, t[s] accounts for the time to load an attribute of the previous version, t[rep] which is equal to 2*t[cpu]+t[s] accounts for the time of replacing an attribute, t[mov] accounts for the time required to move a historical version of an object instance within the memory,

and Avg accounts for the average number of versions of an object instance needs to be materialized in a TDB. In t[rep], the 2*t[cpu] accounts for the time to decode the attribute name and the time to check whether it is the end of a record, and t[s] accounts for the time to replace the data of its subsequent version.

* time required to materialize temporal data of a TDB:
t[mat_S_obj] = S * t[mat_a_obj];

t[o_TDB] = 2 * t[access] + t[read_TDB] + t[mat_S_obj];

2. Materialize the entire history of an object class

t[o_history] = TB * t[o_TDB];

F.2 Tuple time-stamping

Materialize the temporal tuples of a TDB duration

Since temporal data are organized in chronological order in an append-only database, the relevant temporal data are stored in consecutive physical storage areas. In this case, the system only need to read those relevant temporal data into main memory and compare the time notions of the temporal tuples with the specified period.

t[tu_TDB] = t[access] + t[tu_read_history] + t[tu_process_TDB];
t[tu_read_history] = ceil(S*(N̄+2)*A*Avg/Pg) * t[I/O];
t[tu_process_TDB] = S*Avg*(2*(2*t[s]+t[cpu])+t[mov]);

where 2 * t[s] accounts for the time to load a time stamp from a temporal tuple and the predicate for comparison, t[cpu] accounts for the time to compare two time notions, 2 in front of parenthesis accounts for the fact that each temporal tuple contains a start-time and an end-time, and t[mov] accounts for moving a tuple within memory.

Materialize the entire history of a relation

t[tu_history] = t[access] + t[tu_read_history];

F.3 Attribute time-stamping

To preserve efficiency for temporal join operation in materializing temporal data, we pre-sort the TNF relations with the Temporal-Sort-Merge-Join algorithm which consists of a "two-phases sorting" [BLA77] and a "temporal join" for two temporal relations over a certain time period. In the following, we show only the temporal join part of this algorithm which will be used in computing the cost of the third step of materializing temporal data:

```
Algorithm Temporal-Join:
1. set R to be empty;
2. read one tuple r1 from R1 and one tuple r2 from R2;
/* R1 and R2 are the binary relations to be joined */
3. if S(r1) = S(r2)
   /* S() is a function which returns the surrogate (i.e.,key) of a tuple */
        { if ((START(r1) ≥ START(r2) and START(r1) ≤ END(r2)))
          or (START(r2) ≥ START(r1) and START(r2) ≤ END(r1)))
          /*    check if the two time intervals intersect, where START() and
                END() are functions which returns the start-time and end-time of
                a temporal record, respectively   */
                {join r1 and r2 and assign the result to r;
                 START(r) = max(START(r1), START(r2));
                 END(r) = min(END(r1), END(r2));
                 move the joined tuple r to the relation R; }
          else if (START(r1) > END(r2)
                { if (not EOF(R2))
                        {read next tuple from R2 into r2;
                         goto step 3;}
                  else   goto step 6; }
          else
                { if (not EOF(R1))
                        {read next tuple from R1 into r1;
                        goto step 3;}
                  else  goto step 6; }
4.  else if S(r1) > S(r2)
        { if (not EOF(R2))
                {read next tuple from R2 into r2;
                goto step 3;}
          else  goto step 6; }
5. else
        { if (not EOF(R1))
                (read next tuple from R1 into r1;
                goto step 3;}
          else  goto step 6; }
6. End
```

Since the storage model for TNF relations is also append-only, the temporal data are sorted naturally in a chronlogical order and the relevant temporal data in each TNF relation thus are in physically adjacent areas. The cost of materializing temporal data in the attribute time-stamping thus consist of (1) the cost of retrieving the temporal tuples which fall into the specified time interval, (2) the cost of CPU time for sorting the selected temporal tuples of a binary relation based on surrogates (or IIDs) and time tags, (3) the cost of performing temporal joining operations among the temporal tuples.

Materialize the temporal tuples of a TDB duration

* the cost incurred in step 1, i.e., retrieving the relevant data, is t[sl_wt_b_rels]:

    * time required to select temporal tuples from a binary relation:
    t[sl_tpl_b_rel] = t[access] + t[rd_b_rel];
    t[rd_b_rel] = ceil(S*4*A*P*Avg/Pg)*(t[I/O]+ 2* (2*t[s]+t[cpu]) + t[mov_b]);

    where, t[access] accounts for the time to locate the binary relation, t[rd_b_rel] accounts for the number of disk I/Os to read the pages of a binary relation and for the CPU time to compare the time stamps of each tuple with the specified period, 2*t[s] accounts for the time to load two time values to be compared, t[cpu] accounts for the time to perform a comparison (either Start time or End time of a tuple) and t[mov_b] accounts for the time to move a binary tuple to the buffer for transfer to the disk.

    * time required to store the selected intermediate result:
    t[wt_b_rel] = t[access] + ceil(S*4*A*P*Avg/Pg)*t[I/O];

    * time required to select relevant tuples from N-1 binary relations:
    t[sl_tpl_rels] = (N-1) * t[sl_tpl_b_rel];
    * time required to write the N-1 intermediate results:
    t[wt_b_rels] = (N-1) * t[wt_b_rel];

    * total time required to select and write the intermediate result:
    t[sl_wt_b_rels] = t[sl_tpl_rels] + t[wt_b_rels];

* the cost incurred in step 2 is t[st_b_rels] as derived below:

Let m = ceil (S*4*A*Avg*P/Pg); /*the # of pages in an intermediate binary relation*/
and t = Pg/(4*A); /* the number of binary tuples in a page */

* time required to read and write m pages of an intermediate relation:
t[rd_wt_m] = 2 * (t[access] + m * t[I/O]);

* time required to perform (m/2) two-page Merge operations:
t[merge_2_pg] = (m/2) * 2 * t * (2*t[s] + t[cpu] + t[mov_b]);
= m * t * (2*t[s] + t[cpu] + t[mov_b]);

where 2 * t[s] accounts for the loading of two attribute values into registers, t[cpu] is the time for a comparison, and t[mov_b] is the time for moving one binary tuple within main memory.

* time required to implement one merge-sort subphase:
t[1_mg_st] = t[rd_wt_m] + t[merge_2_pg];

For $\log_4(m)$ subphases, the time required to sort a relation R:

* time required to sort one binary relation:
t[st_b_rel] = $\log_4(m)$ * t[1_mg_st];

* time required to sort N-1 binary relations:
t[st_b_rels] = (N-1) * t[st_b_rel];

* the cost incurred in step 3 is t[join_TDB] as derived below:

Since each binary relation with the duration of a TDB has an average size of m pages, the time to read each intermediate binary relation will be t[access] + m * t[I/O]. For a surrogate S1 which has M evolutions in relation R1 and N evolutions in relation R2, the temporal join for this surrogate S1 between R1 and R2 will produce a maximum of (N + M) tuples and a minimum of max(N, M) tuples. Therefore, the total number of generated tuples for each surrogate after a temporal join between two binary relations each having an average of Avg * P evolutions will be:

(1) maximum number of generated tuples = 2*Avg*P
/* when the Start-time and End-time in two relations are all different */
(2) minimum number of generated tuples = max(Avg*P, Avg*P)
/* when both relations have the same Start and End times for all the tuples */

We shall simply use the arithmetic mean of the maximum (M+N) and the minimum MAX(M,N) as the total number of generated tuples for each surrogate in a temporal join, i.e., (3/2) * Avg * P. One limitation, however, has to be obeyed in each temporal

join operation: the generated temporal tuples for each surrogate should not exceed the average number of its evolution (i.e., (3/2) * Avg * P < = Avg). In the case when this relationship does not hold, Avg should be used as the number of generated tuples from a temporal join. For simplicity of representation, we use the following notations in the derivations of analytical formulas:

MT_i = min(½*Avg*P+MT_{i-1}, Avg);
MT_0 = P * Avg;

where, ½*P*Avg accounts for the average increase of generated tuples in each temporal join operation.

* time required to read pages from each intermediate binary relation:
t[rd_i_b_rel] = m * t[I/O];

* time to access each intermediate binary relation:
t[access_b_rel] = t[access] + t[rd_i_b_rel];

* time required to perform temporal join between the first two binary relations:
let Ce = 5 * (2*t[s]+t[cpu]);
Se = 2 * (2*t[s]+t[cpu]);

t[join_1] = S * MT_1 * (Ce + Se + (5/4) * t[mov_b]);

where, MT_1=min(½*P*Avg+MT_0, Avg), Ce=5*(2*t[s]+t[cpu]) accounts for the time to evaluate the conditions stated in step 3 of the Temporal Join algorithm (i.e., "if S(r1) = S(r2)", "if Start(r1) ≥ Start(r2)", "if Start(r1) ≤ End(r2)", "if Start(r2) ≥ Start(r1)", and "if Start(r2) ≤ End(r1)"), Se = 2*(2*t[s]+t[cpu]) accounts for the time to evaluate "Start(r) = max(Start(r1), Start(r2))" and "End(r) = min(End(r1), End(r2))", and (5/4) * t[mov_b] accounts for the time to move a joined tuple which contains one time-invariant, two time-variant attributes and two time stamps to the buffer for transfer to the disk.

* time required to write the intermediate result R onto disk:
t[write_R_1] = t[access] + ceil(S*MT_1*5*A/Pg) * t[I/O];

where, 5 accounts for the fact that each joined tuple contains 3 attributes and two time stamps.

* time required to join the first two binary relation:
t[temp_join_1] = 2 * t[access_b_rel] + t[join_1] + t[write_R_1];

* time required to join the third relation with the intermediate result:
t[read_R_1] = t[write_R_1];
t[join_2] = S * MT_2 * (Ce + Se + (6/4) * t[mov_b]),
where MT_2=min(½*Avg*P+MT_1, Avg);

t[write_R_2] = t[access] + ceil(S*MT_2*6*A/Pg) * t[I/O];
therefore,
t[temp_join_2] = t[access_b_rel] + t[read_R_1] + t[join_2] + t[write_R_2];

* time required to join the fourth relation with the intermediate result,
t[read_R_2] = t[write_R_2];
t[join_3] = S * MT_3 * (Ce + Se + (7/4) * t[mov_b]),
where MT_3=min(½*Avg*P+MT_2, Avg);

t[write_R_3] = t[access] + ceil(S*MT_3*7*A]/Pg) * t[I/O];
therefore,
t[temp_join_3] = t[access_b_rel] + t[read_R_2] + t[join_3] + t[write_R_3];
.
.
* time required to join the (K+1)th relation, where K > 1, with previous result:
t[read_R_K-1] = t[access] + ceil(S*MT_{K-1}*(K+3)*A/Pg) * t[I/O];
t[join_K] = S * MT_K * (Ce + Se + ((K+4)/4) * t[mov_b]);
t[write_R_K] = t[access] + ceil(S*MT_K*(K+4)*A/Pg) * t[I/O];
therefore,
t[temp_join_K] = t[access_b_rel] + t[read_R_K-1] + t[join_K]+ t[write_R_K];

* time required to materialize a temporal relation within TDB duration:
t[join_TDB]
= Σ^i t[temp_join_i];
= (N-1) * t[access_b_rel] + Σ^i t[join_i] + Σ^j (t[read_R_j] + t[write_R_j]);
= (N-1) * t[access_b_rel] + Σ^i t[join_i] + 2 * Σ^j (t[write_R_j]);

where i is the integer ranging from 1 to N-2, and j is the integer ranging from 1 to N-3.

* time required to materialize the temporal tuples of an equivalent TDB:

t[a_TDB] = t[sl_wt_b_rels] + t[st_b_rels] + t[join_TDB];

Materialize the entire history of a relation

In this case, since all the temporal data of a binary relation qualify, there will be only costs of sorting and joining the temporal tuples of the N-1 binary relations.

* time required to sort a binary relation of the entire history:
let mh = ceil(S*4*A*N_{avg}*P/Pg);

* time required to read and write mh pages of a relation:
t[rd_wt_mh] = 2 * (t[access] + mh * t[I/O]);

* time necessary to perform (mh/2) two-page Merge operations:
t[mg_2pg] = mh * t * (2*t[s] + t[cpu] + t[mov_b]);

where 2 * t[s] accounts for the loading of two attribute values into registers, t[cpu] is the time for a comparison, and t[mov_b] is the time for moving a tuple of a binary relation in main memory.

* time required to implement one merge-sort subphase:
t[mg_st] = t[rd_wt_mh] + t[mg_2pg];
for $\log_4(mh)$ subphases, the time required to sort a relation R:

* time required to sort one binary relation:
t[a_st_rel] = $\log_4(mh)$ * t[mg_st];

* time required to sort N-1 binary relations:
t[a_st_rels] = (N-1) * t[a_st_rel]

* time required to join two binary relations:
t[a_join_i] = S * MT_h_i * (Ce + Se + ((K+4)/4) * t[mov_b]);
t[wt_R_j] = t[access] + ceil(S*MT_h_j*(j+4)*A/Pg)*t[I/O],

where $MT_h_i = TB*MT_i$ and $MT_h_0 = TB*MT_0$;

t[access_b_rels] = t[access] + ceil(S*4*A*P*N_{avg}/Pg) * t[I/O];
t[a_join_rels] = (N-1)*t[access_b_rels] + Σ^i t[a_join_i] + 2 * Σ^j (t[wt_R_j]),

where i is the integers from 1 to N-2 and j is the integers from 1 to N-3;

* time required to materialize the entire history:
t[a_h] = t[a_st_rels] + t[a_join_rels];

APPENDIX G
ANALYTICAL FORMULAS FOR TEMPORAL QUERY PROCESSING

G.1 Query type 1

(1) $C[I_1,1] = H_s + \frac{1}{2} * (1 + L_{TDB})$, where H_s (= $ceil(\log_f(S))$) accounts for the cost of searching the instance identifier (IID) tree of a TDB for the desired object instance, 1 in the parenthesis accounts for the I/O needed to access data in the best case, L_{TDB} (= $ceil((N+2)*A/Pg) + ceil((M+2)*A*Avg/Pg)$) accounts for the number of disk I/Os to access data by traversing the history chains through the pvp pointers in a TDB duration in the worst case, and $\frac{1}{2}$ accounts for the average of the best and the worst cases.

(2) $C[I_2,1] = H_s + \frac{1}{2} * (1 + L_T) + 1$, where H_s accounts for the cost of searching the IID tree for the desired object instance, 1 accounts for the I/O to access the pointer of the desired data in the best case, L_T (= $ceil(N_{avg}/f)$) accounts for the number of disk I/Os to traverse the Accession List of an object instance, $\frac{1}{2}$ accounts for the average of the best and the worst cases, and the other 1 accounts for the disk I/O for accessing data.

(3) $C[I_3,1] = H_t + \frac{1}{2}*(1+ceil(S/f)) + 1$, where H_t (=$ceil(\log_f(S*(N_{avg}+1)/S_{avg}))$) accounts for the number of disk I/Os to search the Time Index for the qualified index time point, $\frac{1}{2}*(1+ceil(S/f))$ accounts for the average number of disk I/Os to select the desired pointer from the S pointers of the indexed bucket, and 1 accounts for the disk I/O to access the desired version of a record. Since each of the S object instances has

N_{avg} evolutions, $N_{avg}+1$ index time points will be created for each record during a T duration.

(4) $C[I_4,1] = H_s + H_{ts} + 1$, where H_s accounts for the search cost of the IID tree, H_{ts} accounts for the search cost of the local Time Index of an object instance, and 1 accounts for the access of data. Each object instance in this case has N_{avg} evolution and the number of time points in the local Time Index for each object instance is $N_{avg}+1$.

(5) $C[I_5,1] = H_s + H_{ap} + 1$, where Hs accounts for the search cost of the IID tree, H_{ap} accounts for the search cost of the local AP-tree of an object instance, and 1 accounts for the access of data. Since each object instance (or record) has N_{avg} evolution, each AP-tree has N_{avg} time points.

G.2 Query type 2

For this query type, we assume that the specified time interval involves n TDBs.

(1) $C[I_1,2] = H_s + n * L_{TDB}$, where H_s accounts for the search cost of the desired object instance in the first qualified TDB of the specified interval, $n * L_{TDB}$ accounts for the I/Os to read all the versions of an object instance of n TDBs following the version pointers. Note, in this case, the versions of an object instance within the relevant TDBs can be found using the pvp pointers once the object instance in the first (or the latest) TDB of the specified interval is located. Therefore, the search operation represented by H_s needs to be done only once for n TDBs where n is greater than or equal to one.

(2) $C[I_2,2] = H_s + \frac{1}{2}*(1+L_T) + \frac{1}{2}*(n*Avg + ceil(n*Avg*(N+2)*A/Pg))$, where H_s accounts for the search cost of the IID tree, $\frac{1}{2}*(1+L_T)$ accounts for the average number of disk I/Os to traverse the Accession List of an object instance for locating the pointer of the first qualified historical version, and $\frac{1}{2}*(n*Avg + ceil(n*Avg*(N+2)*A/Pg))$ accounts for the average number of disk I/Os to retrieve the temporal data.

(3) $C[I_3,2] = H_t + \frac{1}{2}*(1+ceil(S/f)) + \frac{1}{2}*(n*Avg + ceil(n*Avg*(N+2)*A/Pg))$, where H_t accounts for the search cost of the Time Index for the first qualified leaf node, $\frac{1}{2}*(1+ceil(S/f))$ accounts for the average I/Os to locate the pointer of the desired object instance from the S pointers of the indexed bucket, and $\frac{1}{2}*(n*Avg + ceil(n*Avg*(N+2)*A/Pg))$ accounts for the average number of disk I/Os to access the temporal data by following the pvp pointers.

(4) $C[I_4,2] = H_s + H_{ts} + \frac{1}{2}*(n*Avg + ceil(n*Avg*(N+2)*A/Pg))$, where H_s accounts for the search cost of the IID tree for the desired object instance, H_{ts} accounts for the search cost of the local Time Index to access the pointer of the first qualified historical version, and $\frac{1}{2}*(n*Avg + ceil(n*Avg*(N+2)*A/Pg))$ accounts for the average number of disk I/Os to access the temporal data by following the pvp pointers.

(5) $C[I_5,2] = H_s + H_{ap} + \frac{1}{2}*(n*Avg + ceil(n*Avg*(N+2)*A/Pg))$, where H_s accounts for the search cost of the IID tree for the desired object instance, H_{ap} accounts for the search cost of the local AP-tree to access the pointer of the first qualified historical version, and $\frac{1}{2}*(n*Avg + ceil(n*Avg*(N+2)*A/Pg))$ accounts for the average number of disk I/Os to access the temporal data by following the nvp pointers.

G.3 Query type 3

(1) $C[I_1,3] = ceil(S/f) + S * \frac{1}{2} * (1+L_{TDB})$, where $ceil(S/f)$ accounts for the cost of sequential search of the leaf nodes of the IID tree, and $S * \frac{1}{2} * (1+L_{TDB})$ accounts for the cost of retrieving a particular version of an object instance for S object instances.

(2) $C[I_2,3] = ceil(S/f) + S*\frac{1}{2}*(1+L_T) + \frac{1}{2}*(S*1 + ceil(S*(N+2)*A/Pg))$, where $ceil(S/f)$ accounts for the cost of a sequential search for the leaf nodes of the IID tree, $S*\frac{1}{2}*(1+L_T)$ accounts for the cost of traversing the S Accession Lists for the pointers of the desired temporal object instances, and $\frac{1}{2}*(S*1 + ceil(S*(N+2)*A/Pg))$ accounts for

the average number of disk I/Os needed to retrieve the historical versions of all the object instance at a given time point.

(3) $C[I_3,3] = H_t + ceil(S/f) + \frac{1}{2}*(S*1 + ceil(S*(N+2)*A/Pg))$, where, H_t accounts for the searching cost of the Time Index, ceil(S/f) accounts for the number of disk I/Os to sequentially access the bucket of S pointers of the indexed time point, and $\frac{1}{2}*(S*1 + ceil(S*(N+2)*A/Pg))$ accounts for the average number of disk I/Os to access the temporal data.

(4) $C[I_4,3] = ceil(S/f) + S * H_{ts} + \frac{1}{2}*(S*1 + ceil(S*(N+2)*A/Pg))$, where ceil(S/f) accounts for the cost of a sequential search for the leaf nodes of the IID tree, $S * H_{ts}$ accounts for the cost of retrieving a pointer of a particular historical version for S object instances from S local Time Indices, and $\frac{1}{2}*(S*1 + ceil(S*(N+2)*A/Pg))$ accounts for the average number of disk I/Os to access the temporal data of S object instances.

(5) $C[I_5,3] = ceil(S/f) + S * H_{ap} + \frac{1}{2}*(S*1 + ceil(S*(N+2)*A/Pg))$, where ceil(S/f) accounts for the cost of a sequential search for the leaf nodes of the IID tree, $S * H_{ap}$ accounts for the cost of retrieving a pointer of a particular historical version for S object instances from S local AP-trees, and $\frac{1}{2}*(S*1 + ceil(S*(N+2)*A/Pg))$ accounts for the average number of disk I/Os to access the temporal data of S object instances.

G.4 Query type 4

(1) $C[I_1,4] = n * (ceil(S*A*(N+2)/Pg) + ceil(S*A*(M+2)*Avg/Pg))$, where n accounts for the number of TDBs involved in the query, ceil(S*A*(N+2)/Pg) accounts for the number of disk I/Os to retrieve data from the RDSA of a TDB, and ceil(S*A*(M+2)*Avg/Pg) accounts for the number of disk I/Os to retrieve data from the HDSA of a TDB. In this case, since all the temporal data of a TDB will be retrieved there is no need to search the index and the access of data can start with the beginning of a TDB.

(2) $C[I_2,4] = ceil(S/f) + S*[\frac{1}{2}*(1+L_T) + \frac{1}{2}*(n*Avg+ceil(n*(N+2)*A*Avg/Pg))]$, where ceil(S/f) accounts for the number of disk I/Os to search sequentially the leaf nodes of the IID tree, S accounts for the fact that S object instances are retrieved, $\frac{1}{2}*(1+L_T)$ accounts for the average number of disk I/Os to traverse the Accession List of the historical versions of an object instance to locate the first qualified version, and $\frac{1}{2}$ * (n*Avg + ceil(n * (N+2) * A * Avg/Pg)) accounts for the average number of disk I/Os to retrieve the required versions of an object instance.

(3) $C[I_3,4] = H_t + ceil(S/f) + S * \frac{1}{2}*(n*Avg + ceil(n*(N+2)*A*Avg/Pg))$, where H_t accounts for the search cost of the Time Index for the first qualified leaf node, ceil(S/f) accounts for the number of disk I/Os needed to read all the S pointers in the indexed bucket, and $S * \frac{1}{2}*(n*Avg + ceil(n*(N+2)*A*Avg/Pg))$ accounts for the number of disk I/Os in the average case to access the data from the S pointers of the indexed bucket.

(4) $C[I_4,4] = ceil(S/f) + S * (H_{ts} + \frac{1}{2}*(n*Avg + ceil(n*(N+2)*A*Avg/Pg)))$, where ceil(S/f) accounts for the cost of a sequential search for the S object instances in the IID tree, H_{ts} accounts for the search cost of a local Time Index to access the first qualified historical version, and $\frac{1}{2}*(n*Avg + ceil(n*(N+2)*A*Avg/Pg))$ accounts for the number of disk I/Os in the average case to access the temporal data of an object instance by following the pvp pointers, and S in S * (...) accounts for the fact that there are S object instances to be retrieved.

(5) $C[I_5,4] = ceil(S/f) + S * (H_{ap} + \frac{1}{2}*(n*Avg + ceil(n*(N+2)*A*Avg/Pg)))$, where ceil(S/f) accounts for the cost of a sequential search for the S object instances in the IID tree, H_{ap} accounts for the search cost of a local AP-tree to access the first qualified version, and $\frac{1}{2}*(n*Avg + ceil(n*(N+2)*A*Avg/Pg))$ accounts for the number of disk I/Os in the average case to access the temporal data of an object instance by

following the pvp pointers, and S in S * (...) accounts for the fact that there are S object instances to be retrieved.

G.5 Query type 5

(1) $C[I_1,5] = H_s + TB*L_{TDB}$, where H_s accounts for the number of disk I/Os to search for a particular object instance from the most recent TDB containing current information and $TB*L_{TDB}$ accounts for the number of disk I/Os to read all the versions of an object instance of all TDBs following pvp pointers.

(2) $C[I_2,5] = H_s + \frac{1}{2}*(N_{avg} + ceil(N_{avg}*(N+2)*A/Pg))$, where H_s accounts for the number of disk I/Os to search the IID tree for the object instance, and $\frac{1}{2}*(N_{avg} + ceil(N_{avg}*(N+2)*A/Pg))$ accounts for the number of disk I/Os in the average case to retrieve the history of an object instance by following the pvp pointers.

(3) $C[I_3,5] = \frac{1}{2}*(1+ceil(S/f)) + \frac{1}{2}*(N_{avg} + ceil(N_{avg}*(N+2)*A/Pg))$, where $\frac{1}{2}*(1+ceil(S/f))$ accounts for the average number of disk I/Os needed to search the S pointers in the indexed bucket of the latest time point for the desired object instance and $\frac{1}{2}*(N_{avg} + ceil(N_{avg}*(N+2)*A/Pg))$ accounts for the average number of disk I/Os needed to access the history of an object instance by following the pvp pointers.

(4) $C[I_4,5] = H_s + \frac{1}{2}*(N_{avg} + ceil(N_{avg}*(N+2)*A/Pg))$, where H_s accounts for the search cost of the IID tree for the desired object instance and $\frac{1}{2}*(N_{avg} + ceil(N_{avg}*(N+2)*A/Pg))$ accounts for the average number of disk I/Os needed to access the history of an object instance by following the pvp pointers.

(5) $C[I_5,5] = H_s + \frac{1}{2}*(N_{avg} + ceil(N_{avg}*(N+2)*A/Pg))$, where, H_s accounts for the search cost of the IID tree for the desired object instance and $\frac{1}{2}*(N_{avg} + ceil(N_{avg}*(N+2)*A/Pg))$ accounts for the average number of disk I/Os needed to access the history of an object instance by following the nvp pointers.

G.6 Query type 6

(1) $C[I_1,6] = H_s + 1$.

(2) $C[I_2,6] = H_s + 1$.

(3) $C[I_3,6] = 2 + \frac{1}{2} * (1+\text{ceil}(S/f)) + 1$, where, 2 accounts for the number of disk I/Os to access the root and the right most leaf nodes of the Time Index, $\frac{1}{2}*(1+\text{ceil}(S/f))$ accounts for the average number of disk I/Os to search the S pointers of the indexed bucket of current time, and 1 accounts for the access of the current data of a surrogate from the pointer in the indexed bucket.

(4) $C[I_4,6] = H_s + 2 + 1$, where, H_s accounts for the search cost of the IID tree, 2 accounts for the number of disk I/Os to access the root and the right most leaf nodes of the local Time Index, and 1 accounts for the access to the data.

(5) $C[I_5,6] = H_s + H_{ap} + 1$, where, H_s accounts for the search cost of the IID tree, 2 accounts for the number of disk I/Os to access the root and the right most leaf nodes of the local AP-tree, and 1 accounts for the access to the data.

G.7 Query type 7

(1) $C[I_1,7] = 1 + \text{ceil}(S*(N+2)*A/Pg)$, where 1 accounts for the disk I/O to access the pointer to the beginning of current data and $\text{ceil}(S*(N+2)*A/Pg)$ accounts for the number of disk I/Os to read sequentially the current versions of all the object instances.

(2) $C[I_2,7] = 1 + \text{ceil}(S*(N+2)*A/Pg)$.

(3) $C[I_3,7] = 1 + \text{ceil}(S*(N+2)*A/Pg)$.

(4) $C[I_4,7] = 1 + \text{ceil}(S*(N+2)*A/Pg)$.

(5) $C[I_5,7] = 1 + \text{ceil}(S*(N+2)*A/Pg)$.

Note: If the current data is not separated from the historical data in the storage models of I_3 through I_5, then extra disk I/Os will be needed in these index techniques and their performances will be much worse than I_1 and I_2.

G.8 Query type 8

For this and the following query types, we assume that (1) the total number of distinct values of an indexed attribute in the database at ta given time is NA, (2) each of the NA values has the same number of evolutions EV which are uniformly distributed along the time dimension where $EV=(S*N_{avg}*M/((N-1)*NA))$ and $(S*N_{avg}*M/(N-1))$ is the total number of evolutions for one attribute of an object class and (3) each of the NA values has an average of k instantiations of object instances at each time point. Based on these assumptions, each of the NA attribute values has an average of EV/TB evolutions in a TDB duration. The cost of retrieving the current data which satisfy a predicate is as follows:

(1) $C[I_1,8]=\log_f(NA)+\log_f(EV/TB)+ceil(k/f)+\frac{1}{2}*(k+ceil(k*(N+2)*A/Pg)$, where $\log_f(NA)$ accounts for the number of disk I/Os required to search a local attribute index tree for the desired attribute value, $\log_f(EV/TB)$ accounts for the number of disk I/Os for searching a local time index, ceil(k/f) accounts for the number of disk I/Os for retrieving the pointers of the k historical data sequentially, and $\frac{1}{2}$ * (k + ceil(k*(N+2)*A/Pg) accounts for the average number of disk I/Os to retrieve the qualified data.

(2) $C[I_2,8] = \log_f(NA) + ceil(k/f) + \frac{1}{2}*(k+ceil(k*(N+2)*A/Pg)$, where $\log_f(NA)$ accounts for the number of disk I/Os required to search a local attribute index tree for the desired attribute value, ceil(k/f) accounts for the number of disk I/Os for retrieving the pointers of the k historical data sequentially, and $\frac{1}{2}$ * (k + ceil(k*(N+2)*A/Pg)

accounts for the average number of disk I/Os to retrieve the qualified data. We note here that the Accession List only indexes on the attribute values of current data.

(3) $C[I_3,8] = H_t + ceil(S/f) + \frac{1}{2}*(k+ceil(k*(N+2)*A/Pg)$, where H_t accounts for the number of disk I/Os of searching the global Time Index for the begining of current data, ceil(S/f) accounts for the number of disk I/Os for retrieving the pointers of all current data and $\frac{1}{2}*(k+ceil(k*(N+2)*A/Pg)$ accounts for the average number of disk I/Os to retrieve all the current data.

(4) $C[I_4,8] = \log_f(NA) + \log_f(EV) + ceil(k/f) + \frac{1}{2}*(k+ceil(k*(N+2)*A/Pg)$, where $\log_f(NA)$ accounts for the number of disk I/Os required to search a global attribute index tree for the desired attribute value, $\log_f(EV)$ accounts for the number of disk I/Os for searching a second-level time index, ceil(k/f) accounts for the number of disk I/Os for retrieving the pointers of the k historical data sequentially, and $\frac{1}{2}$ * (k + ceil(k*(N+2)*A/Pg) accounts for the average number of disk I/Os to retrieve the qualified data.

(5) $C[I_5,8]$[1] $= \log_f(NA) + \log_f(EV) + ceil(k/f) + \frac{1}{2}*(k+ceil(k*(N+2)*A/Pg)$, where $\log_f(NA)$ accounts for the number of disk I/Os required to search a global attribute index tree for the desired attribute value, $\log_f(EV)$ accounts for the number of disk I/Os for searching a local time index, ceil(k/f) accounts for the number of disk I/Os for retrieving the pointers of the k historical data sequentially, and $\frac{1}{2}$ * (k+ceil(k*(N+2)*A /Pg) accounts for the average number of disk I/Os to retrieve the qualified data.

---

[1]For this and the following query types, we assume that I_5 picks the "start time" of each evolution of an attribute value as the entries for its second-level time index.

G.9 Query type 9

(1) $C[I_1,9] = \log_f(NA)+\log_f(EV/TB)+ceil(k/f)+ k*\frac{1}{2}*(1+L_{TDB})$, where $\log_f(NA/TB)$ accounts for the number of disk I/Os required to search a local attribute index tree for the desired attribute value, $\log_f(EV/TB)$ accounts for the number of disk I/Os for searching a local time index, ceil(k/f) accounts for the number of disk I/Os for retrieving the pointers of the k historical data sequentially, and $k*\frac{1}{2}*(1+L_{TDB})$ accounts for the average number of disk I/Os to retrieve the qualified data.

(2) $C[I_2,9] = ceil(S/f) + S*\frac{1}{2}*(1+L_{T)} + \frac{1}{2}*(k+ceil(k*(N+2)*A/Pg)$, where ceil(S/f) accounts for the number of disk I/Os to sequentially access the current versions of the S object instances, $S*\frac{1}{2}*(1+L_T)$ accounts for the average number of disk I/Os to search the historical chains of the S object instances for the qualified data and $\frac{1}{2}*(k+ceil(k*(N+2)*A/Pg)$ accounts for the average number of disk I/Os to retrieve the qualified data.

(3) $C[I_3,9] = H_t + ceil(S/f) + \frac{1}{2}*(S+ceil(S*(N+2)*A/Pg))$, where H_t accounts for the number of disk I/Os of searching the global Time Index for the designated time point, ceil(S/f) accounts for the number of disk I/Os for retrieving the pointers of all the S object instances at the designated time point and $\frac{1}{2}$ * (S + ceil(S*(N+2)*A/Pg)) accounts for the average number of disk I/Os to retrieve the historical data of all the S object instances at the designated time.

(4) $C[I_4,9] = \log_f(NA) + \log_f(EV) + ceil(k/f) + \frac{1}{2}*(k+ceil(k*(N+2)*A/Pg)$, where $\log_f(NA)$ accounts for the number of disk I/Os required to search a global attribute index tree for the desired attribute value, $\log_f(EV)$ accounts for the number of disk I/Os for searching a second-level time index, ceil(k/f) accounts for the number of disk I/Os for retrieving the pointers of the k historical data sequentially, and $\frac{1}{2}$ * (k +

ceil(k*(N+2)*A/Pg)) accounts for the average number of disk I/Os to retrieve the qualified data.

(5) $C[I_5,9] = \log_f(NA)+\log_f(EV))+ceil(k/f)+\frac{1}{2}*(k+ceil(k*(N+2)*A/Pg))$, where $\log_f(NA)$ accounts for the number of disk I/Os required to search a global attribute index tree for the desired attribute value, $\log_f(EV)$ accounts for the number of disk I/Os for searching a local time index, ceil(k/f) accounts for the number of disk I/Os for retrieving the pointers of the k historical data sequentially, and $\frac{1}{2}$ * (k + ceil(k*(N+2)*A /Pg)) accounts for the average number of disk I/Os to retrieve the qualified data.

G.10 Query type 10

For this query type, we assume that the specified interval involves a total of n TDBs and the total number of qualified object instances is n*k. Based on this assumption, we have the following derivation:

(1) $C[I_1,10]=n*(\log_f(NA)+\log_f(EV/TB)+ceil(k/f)+k*\frac{1}{2}*(1+L_{TDB}))$, where n in $n*(\log_f(NA)+...)$ accounts for the number of TDBs involved in the specified time interval, $\log_f(NA)$ accounts for the number of disk I/Os required to search a local attribute index tree for the desired attribute value, $\log_f(EV/TB)$ accounts for the number of disk I/Os for searching a local time index, ceil(k/f) accounts for the number of disk I/Os for retrieving the pointers of the k historical data sequentially, and $k*\frac{1}{2}*(1+L_{TDB})$ accounts for the average number of disk I/Os to retrieve the qualified data.

(2) $C[I_2,10] = ceil(S/f) + S*\frac{1}{2}*(1+L_T) + \frac{1}{2}*(n*k+ceil(n*k*(N+2)*A/Pg)$, where ceil(S/f) accounts for the number of disk I/Os to sequentially access the current versions of the S object instances, $S*\frac{1}{2}*(1+L_T)$ accounts for the average number of disk I/Os to search the historical chains of the S object instances for the qualified data and $\frac{1}{2}*(n*k+ceil(n*k*(N+2)*A/Pg)$ accounts for the average number of disk I/Os to retrieve the qualified data.

(3) $C[I_3,10] = H_t + ceil(S/f) + S*\frac{1}{2}*(n*Avg + ceil(n*(N+2)*A*Avg/Pg))$, where H_t accounts for the search cost of the Time Index for the first qualified leaf node, ceil(S/f) accounts for the number of disk I/Os needed to read all the S pointers in the indexed bucket, and $S * \frac{1}{2}*(n*Avg + ceil(n*(N+2)*A*Avg/Pg))$ accounts for the number of disk I/Os in the average case to access the data from the S pointers of the indexed bucket.

(4) $C[I_4,10] = \log_f(NA)+\log_f(EV)+ceil(n*k/f)+\frac{1}{2}*(n*k+ceil(n*k*(N+2)*A/Pg)$, where $\log_f(NA)$ accounts for the number of disk I/Os required to search a global attribute index tree for the desired attribute value, $\log_f(EV)$ accounts for the number of disk I/Os for searching a second-level time index, ceil(n*k/f) accounts for the number of disk I/Os for retrieving the pointers of the k historical data sequentially, and $\frac{1}{2}*(n*k + ceil(n*k*(N+2)*A/Pg))$ accounts for the average number of disk I/Os to retrieve the qualified data.

(5) $C[I_5,10] =$

$\log_f(NA)+\log_f(EV))+ceil(n*k/f)+\frac{1}{2}*(n*k+ceil(n*k*(N+2)*A/Pg))$, where $\log_f(NA)$ accounts for the number of disk I/Os required to search a global attribute index tree for the desired attribute value, $\log_f(EV)$ accounts for the number of disk I/Os for searching a local time index, ceil(n*k/f) accounts for the number of disk I/Os for retrieving the pointers of the k historical data sequentially, and $\frac{1}{2} * (n*k + ceil(n*k*(N+2)*A/Pg))$ accounts for the average number of disk I/Os to retrieve the qualified data.

APPENDIX H
PERFORMANCE EVALUATION AND COMPARISON OF INDEXING TECHNIQUES

H.1 Query type 1

(1) Vary the average modified attributes from 1 to 41

M=	$C[I_1,1]$	$C[I_2,1]$	$C[I_3,1]$	$C[I_4,1]$	$C[I_5,1]$
1	3.5	4.5	33.5	5	5
5	3.5	4.5	33.5	5	5
9	3.5	4.5	33.5	5	5
13	4	4.5	33.5	5	5
17	4	4.5	33.5	5	5
21	4	4.5	33.5	5	5
25	4.5	4.5	33.5	5	5
29	4.5	4.5	33.5	5	5
33	4.5	4.5	33.5	5	5
37	5	4.5	33.5	5	5
41	5	4.5	33.5	5	5

(2) Vary the average # of evolutions N_{avg} from 100 to 1000 when M=10 and N=20

N_{avg} =	$C[I_1,1]$	$C[I_2,1]$	$C[I_3,1]$	$C[I_4,1]$	$C[I_5,1]$
100	3.5	4	33.5	4	4
200	3.5	4.5	33.5	5	5
300	3.5	4.5	33.5	5	5
400	4	5	33.5	5	5
500	4	5	33.5	5	5
600	4	5.5	33.5	5	5
700	4.5	6	33.5	5	5
800	4.5	6	33.5	5	5
900	4.5	6.5	33.5	5	5
1000	4.5	6.5	33.5	5	5

(3) Vary the page size Pg from 1K to 10K when M=10 and N=20

Pg = (K)	$C[I_1,1]$	$C[I_2,1]$	$C[I_3,1]$	$C[I_4,1]$	$C[I_5,1]$
1	5	6.5	63.5	6	6
2	3.5	4.5	33.5	5	5
3	3.5	4.5	23.5	5	5
4	3.5	4	18.5	4	4
5	3.5	4	15.5	4	4
6	3.5	4	13.5	4	4
7	3.5	4	12.5	4	4
8	3.5	4	11	4	4
9	3.5	4	10.5	4	4
10	3.5	4	9.5	4	4

(4) Vary the number of surrogates from 10000 to 100000 when M=10 and N=20

S=	$C[I_1,1]$	$C[I_2,1]$	$C[I_3,1]$	$C[I_4,1]$	$C[I_5,1]$
10000	3.5	4.5	33.5	5	5
20000	3.5	4.5	63.5	5	5
30000	4.5	5.5	93.5	6	6
40000	4.5	5.5	123.5	6	6
50000	4.5	5.5	153.5	6	6
60000	4.5	5.5	183.5	6	6
70000	4.5	5.5	213.5	6	6
80000	4.5	5.5	243.5	6	6
90000	4.5	5.5	273	6	6
100000	4.5	5.5	304	6	6

(5) Vary S_{avg} from 1 to 5001 when M=10 and N=20

S_{avg} =	$C[I_1,1]$	$C[I_2,1]$	$C[I_3,1]$	$C[I_4,1]$	$C[I_5,1]$
1	3.5	4.5	34.5	5	5
501	3.5	4.5	33.5	5	5
1001	3.5	4.5	33.5	5	5
1501	3.5	4.5	33.5	5	5
2001	3.5	4.5	33.5	5	5
2501	3.5	4.5	33.5	5	5
3001	3.5	4.5	33.5	5	5
3501	3.5	4.5	33.5	5	5
4001	3.5	4.5	33.5	5	5
4501	3.5	4.5	33.5	5	5
5001	3.5	4.5	33.5	5	5

Conclusion of query type 1:

(1) In general, $I_1 < I_2 < I_4 = I_5 < I_3$ for this type of query.

(2) As M increases, only I_1 is affected and deteriorates accordingly. However, I_1 still has the best performance for this type of query unless M is large (e.g., 37 in this case):

when M < 25,	$I_1 < I_2 < I_4 = I_5 < I_3$
when 25 <= M < 37,	$I_1 = I_2 < I_4 = I_5 < I_3$
when 37 < M < 700,	$I_2 < I_4 = I_5 < I_1 < I_3$
when 700 <= M,	$I_2 < I_4 = I_5 < I_3 < I_1$

(3) As N_{avg} increases, so does the required disk I/Os for each indexing technique. The following is the conclusion when M=10 and N=20:

when N_{avg} < 400,	$I_1 < I_2 < I_4 = I_5 < I_3$
when 400 <= N_{avg} < 600	$I_1 < I_2 < I_4 = I_5 < I_3$
when 600 <= N_{avg} < 1100	$I_1 < I_4 = I_5 < I_2 < I_3$
when 1100 <= N_{avg} <1300	$I_1 = I_4 = I_5 < I_2 < I_3$
when 1300 <= N_{avg}	$I_4 = I_5 < I_1 < I_2 < I_3$

(4) As page size Pg increases, the fan-out factor f increases and so does the required number of disk I/Os. The relative performances of the indexing techniques are given as below when M=10 and N=20:

when Pg < 2K,	$I_1 < I_4 = I_5 < I_2 < I_3$
when 2K <= Pg <4K	$I_1 < I_2 < I_4 = I_5 < I_3$
when 4K <= Pg	$I_1 < I_2 = I_4 = I_5 < I_3$

(5) As S increases, the required number of disk I/Os in each indexing technique increases. The relative performances of the indexing techniques thus depend on the other parameters.

(6) S_{avg} affects I3 not significantly. As S_{avg} increases, performance of I_3 improves.

(7) The performance of I_2 is very close to that of I_4 and I_5.

(8) I_4 and I_5 have the same performance; I_3 has the worst performance because it requires a sequential search for the desired pointer from the bucket of pointers of the indexed time point; the performances of the other techniques are very close.

H.2 Query type 2

(1) Vary the average modified attributes M from 1 to 41

M =	C[I_1,2]	C[I_2,2]	C[I_3,2]	C[I_4,2]	C[I_5,2]
1	4	5.5	34.5	6	6
5	4	5.5	34.5	6	6
9	4	5.5	34.5	6	6
13	5	5.5	34.5	6	6
17	5	5.5	34.5	6	6
21	5	5.5	34.5	6	6
25	6	5.5	34.5	6	6
29	6	5.5	34.5	6	6
33	6	5.5	34.5	6	6
37	7	5.5	34.5	6	6
41	7	5.5	34.5	6	6

(2) Vary the number of retrieved TDBs n from 1 to 10 when M=10 and N=20

n =	C[I_1,2]	C[I_2,2]	C[I_3,2]	C[I_4,2]	C[I_5,2]
1	4	5.5	34.5	6	6
2	6	7.5	36.5	8	8
3	8	9.5	38.5	10	10
4	10	10.5	39.5	11	11
5	12	12.5	41.5	13	13
6	14	14.5	43.5	15	15
7	16	15.5	44.5	16	16
8	18	17.5	46.5	18	18
9	20	19.5	48.5	20	20
10	22	20.5	49.5	21	21

(3) Vary the average number of evolutions N_{avg} from 100 to 1000

N_{avg} =	C[I_1,2]	C[I_2,2]	C[I_3,2]	C[I_4,2]	C[I_5,2]
100	4	4	33.5	4	4
200	4	5.5	34.5	6	6
300	4	5.5	34.5	6	6
400	5	7	35.5	7	7
500	5	7	35.5	7	7
600	5	8.5	36.5	8	8
700	6	9	36.5	8	8
800	6	10	37.5	9	9
900	6	11.5	38.5	10	10
1000	6	11.5	38.5	10	10

(4) Vary the page size Pg from 1K to 10K

Pg =	C[I_1,2]	C[I_2,2]	C[I_3,2]	C[I_4,2]	C[I_5,2]
1	6	9.5	66.5	9	9
2	4	5.5	34.5	6	6
3	4	5.5	24.5	6	6
4	4	4	18.5	4	4
5	4	4	15.5	4	4
6	4	4	13.5	4	4
7	4	4	12.5	4	4
8	4	4	11	4	4
9	4	4	10.5	4	4
10	4	4	9.5	4	4

(5) Vary the number of surrogates S from 10000 to 100000

S =	C[I_1,2]	C[I_2,2]	C[I_3,2]	C[I_4,2]	C[I_5,2]
10000	4	5.5	34.5	6	6
20000	4	5.5	64.5	6	6
30000	5	6.5	94.5	7	7
40000	5	6.5	124.5	7	7
50000	5	6.5	154.5	7	7
60000	5	6.5	184.5	7	7
70000	5	6.5	214.5	7	7
80000	5	6.5	244.5	7	7
90000	5	6.5	274	7	7
100000	5	6.5	305	7	7

(6) Vary the average number of modified surrogates at a time S_{avg} from 1 to 5001

S_{avg} =	C[I_1,2]	C[I_2,2]	C[I_3,2]	C[I_4,2]	C[I_5,2]
1	4	5.5	35.5	6	6
501	4	5.5	34.5	6	6
1001	4	5.5	34.5	6	6
1501	4	5.5	34.5	6	6
2001	4	5.5	34.5	6	6
2501	4	5.5	34.5	6	6
3001	4	5.5	34.5	6	6
3501	4	5.5	34.5	6	6
4001	4	5.5	34.5	6	6
4501	4	5.5	34.5	6	6
5001	4	5.5	34.5	6	6

Conclusion of query type 2:

(1) when M < 23, I_1 has the best performance. Because all the required temporal data in the storage model of I_1 are aggregated together in a block, the required number of disk I/Os is greatly reduced. The relative performances of these indexing techniques when M varies are as follows:

when M < 23	$I_1 < I_2 < I_4 = I_5 < I_3$,
when 23 <= M < 25	$I_2 < I_1 < I_4 = I_5 < I_3$,
when 25 <= M < 33	$I_2 < I_1 = I_4 = I_5 < I_3$,
when 33 <= M < 720	$I_2 < I_4 = I_5 < I_1 < I_3$,
when 720 <= M	$I_2 < I_4 = I_5 < I_3 < I_1$,

(2) As n increases (i.e., the specified time interval expands), the processing cost in each approach increases.

when n < 7	$I_1 < I_2 < I_4 = I_5 < I_3$
when 7 <= n <10	$I_2 < I_1 = I_4 = I_5 < I_3$
when 10 <= n	$I_2 < I_4 = I_5 < I_1 < I_3$

(3) As N_{avg} increases, the number of disk I/Os increases. The relative performances of these indexing techniques when M=10 and N= 20 are as follows:

when N_{avg} <= 100,	$I_1 = I_2 = I_4 = I_5 < I_3$
when 100 < Navg < 400,	$I_1 < I_2 < I_4 = I_5 < I_3$
when 400 <= Navg <600,	$I_1 < I_2 = I_4 = I_5 < I_3$
when 600 <= Navg	$I_1 < I_4 = I_5 < I_2 < I_3$

(4) As Pg increase, f increases and so does the required number of disk I/Os.

when Pg < 2K	$I_1 < I_4 = I_5 < I_2 < I_3$
when 2K <= Pg < 4K	$I_1 < I_2 < I_4 = I_5 < I_3$
when 4K <= Pg <= 10K	$I_1 = I_2 = I_4 = I_5 < I_3$

(5) As S increases, the number of disk I/Os in each indexing technique increases. The relative performances of the indexing techniques thus depend on the other parameters.

(6) I_3 has the worst performance because it requires a sequential search for the first qualified pointer from the bucket of pointers of the index time point; performances of the other techniques however are very close.

H.3 Query type 3

(1) Vary the average modified attributes M from 1 to 41

M=	$C[I_1,3]$	$C[I_2,3]$	$C[I_3,3]$	$C[I_4,3]$	$C[I_5,3]$
1	15060	20480	5482	25480	25480
5	15060	20480	5482	25480	25480
9	15060	20480	5482	25480	25480
13	20060	20480	5482	25480	25480
17	20060	20480	5482	25480	25480
21	20060	20480	5482	25480	25480
25	25060	20480	5482	25480	25480
29	25060	20480	5482	25480	25480
33	25060	20480	5482	25480	25480
37	30060	20480	5482	25480	25480
41	30060	20480	5482	25480	25480

(2) Vary the average number of evolutions N_{avg} from 100 to 1000

N_{avg} =	$C[I_1,3]$	$C[I_2,3]$	$C[I_3,3]$	$C[I_4,3]$	$C[I_5,3]$
100	15060	15480	5482	15480	15480
200	15060	20480	5482	25480	25480
300	15060	20480	5482	25480	25480
400	20060	25480	5482	25480	25480
500	20060	25480	5482	25480	25480
600	20060	30480	5482	25480	25480
700	25060	35480	5482	25480	25480
800	25060	35480	5482	25480	25480
900	25060	40480	5482	25480	25480
1000	25060	40480	5482	25480	25480

(3) Vary the page size Pg from 1K to 10K

Pg =	$C[I_1,3]$	$C[I_2,3]$	$C[I_3,3]$	$C[I_4,3]$	$C[I_5,3]$
1	20120	30960.0	5962.0	25960.0	25960.0
2	15060	20480.0	5482.0	25480.0	25480.0
3	15040	20320.0	5322.0	25320.0	25320.0
4	15030	15240.0	5242.0	15240.0	15240.0
5	15024	15192.0	5194.0	15192.0	15192.0
6	15020	15160.0	5162.0	15160.0	15160.0
7	15018	15138.0	5140.0	15138.0	15138.0
8	15015	15120.0	5122.0	15120.0	15120.0
9	15014	15107.5	5109.5	15107.5	15107.5
10	15012	15096.0	5098.0	15096.0	15096.0

(4) Vary the number of surrogates S from 10000 to 100000

S=	$C[I_1,3]$	$C[I_2,3]$	$C[I_3,3]$	$C[I_4,3]$	$C[I_5,3]$
10000	15060	20480	5482	25480	25480
20000	30120	40960	10962	50960	50960
30000	45180	61440	16442	76440	76440
40000	60240	81920	21922	101920	101920
50000	75300	102400	27402	127400	127400
60000	90360	122880	32882	152880	152880
70000	105420	143360	38362	178360	178360
80000	120480	163840	43842	203840	203840
90000	135539	184319	49321	229319	229319
100000	150599	204799	54802	254799	254799

(5) Vary the average number of modified surrogates S_{avg} from 1 to 5001

S_{avg} =	$C[I_1,3]$	$C[I_2,3]$	$C[I_3,3]$	$C[I_4,3]$	$C[I_5,3]$
1	15060	20480	5483	25480	25480
501	15060	20480	5482	25480	25480
1001	15060	20480	5482	25480	25480
1501	15060	20480	5482	25480	25480
2001	15060	20480	5482	25480	25480
2501	15060	20480	5482	25480	25480
3001	15060	20480	5482	25480	25480
3501	15060	20480	5482	25480	25480
4001	15060	20480	5482	25480	25480
4501	15060	20480	5482	25480	25480
5001	15060	20480	5482	25480	25480

Conclusion of query type 3:

(1) In general, I_3 has the best performance in this type of query because pointers of all the required temporal data are aggregated together in a bucket. When M varies, the relative performances among these indexing techniques are as follows:

when $M < 22$,	$I_3 < I_1 < I_2 < I_4 = I_5$
when $22 <= M < 34$	$I_3 < I_2 < I_1 < I_4 = I_5$
when $34 <= M$	$I_3 < I_2 < I_4 = I_5 < I_1$

(2) when $N_{avg} < 400$,	$I_3 < I_1 < I_2 < I_4 = I_5$
when $400 <= N_{avg} < 500$,	$I_3 < I_1 < I_2 = I_4 = I_5$
when $500 <= N_{avg} < 1050$,	$I_3 < I_1 < I_4 = I_5 < I_2$
when $1050 <= N_{avg}$,	$I_3 < I_4 = I_5 < I_1 < I_2$

(3) when $Pg < 2K$	$I_3 < I_1 < I_4 = I_5 < I_2$
when $2K <= Pg < 4K$	$I_3 < I_1 < I_2 < I_4 = I_5$
when $4K <= Pg <= 10K$	$I_3 < I_1 < I_2 = I_4 = I_5$

(4) As S increases, the required number of disk I/Os increases. The relative performances of the indexing techniques depend on the other parameters.

H.4 Query type 4

(1) Vary the average modified attributes from 1 to 41

M=	$C[I_1,4]$	$C[I_2,4]$	$C[I_3,4]$	$C[I_4,4]$	$C[I_5,4]$
1	2040	35060	20062	40060	40060
5	3640	35060	20062	40060	40060
9	5240	35060	20062	40060	40060
13	6840	35060	20062	40060	40060
17	8440	35060	20062	40060	40060
21	10040	35060	20062	40060	40060
25	11640	35060	20062	40060	40060
29	13240	35060	20062	40060	40060
33	14840	35060	20062	40060	40060
37	16440	35060	20062	40060	40060
41	18040	35060	20062	40060	40060

(2) Vary the number of accessed temporal data blocks n from 1 to 10

n =	$C[I_1,4]$	$C[I_2,4]$	$C[I_3,4]$	$C[I_4,4]$	$C[I_5,4]$
1	5640	35060	20062	40060	40060
2	11280	55060	40062	60060	60060
3	16920	75060	60062	80060	80060
4	22560	85060	70062	90060	90060
5	28200	105060	90062	110060	110060
6	33840	125060	110062	130060	130060
7	39480	135060	120062	140060	140060
8	45120	155060	140062	160060	160060
9	50760	175060	160062	180060	180060
10	56400	185060	170062	190060	190060

(3) Vary the average number of evolutions N_{avg} from 100 to 1000

N_{avg} =	$C[I_1,4]$	$C[I_2,4]$	$C[I_3,4]$	$C[I_4,4]$	$C[I_5,4]$
100	2440	20060	10062	20060	20060
200	4040	35060	20062	40060	40060
300	5640	35060	20062	40060	40060
400	7240	50060	30062	50060	50060
500	8841	50060	30062	50060	50060
600	10440	65060	40062	60060	60060
700	12040	70060	40062	60060	60060
800	13640	80060	50062	70060	70060
900	15240	95060	60062	80060	80060
1000	16841	95060	60062	80060	80060

(4) Vary the page size Pg from 1K to 10K

Pg =	$C[I_1,4]$	$C[I_2,4]$	$C[I_3,4]$	$C[I_4,4]$	$C[I_5,4]$
1	11280	65120	40122	60120	60120
2	5640	35060	20062	40060	40060
3	3760	35040	20042	40040	40040
4	2820	20030	10032	20030	20030
5	2256	20024	10026	20024	20024
6	1880	20020	10022	20020	20020
7	1612	20018	10020	20018	20018
8	1410	20015	10017	20015	20015
9	1254	20014	10016	20014	20014
10	1128	20012	10014	20012	20012

(5) Vary the number of surrogates S from 10000 to 100000

S=	$C[I_1,4]$	$C[I_2,4]$	$C[I_3,4]$	$C[I_4,4]$	$C[I_5,4]$
10000	5640	35060	20062	40060	40060
20000	11280	70120	40122	80120	80120
30000	16920	105180	60182	120180	120180
40000	22560	140240	80242	160240	160240
50000	28200	175300	100302	200300	200300
60000	33840	210360	120362	240360	240360
70000	39480	245420	140422	280420	280420
80000	45120	280480	160482	320480	320480
90000	50760	315539	180541	360539	360539
100000	56400	350599	200602	400599	400599

(6) Vary the average number of modified surrogates S_{avg} from 1 to 5001

S_{avg} =	$C[I_1,4]$	$C[I_2,4]$	$C[I_3,4]$	$C[I_4,4]$	$C[I_5,4]$
1	5640	35060	20063	40060	40060
501	5640	35060	20062	40060	40060
1001	5640	35060	20062	40060	40060
1501	5640	35060	20062	40060	40060
2001	5640	35060	20062	40060	40060
2501	5640	35060	20062	40060	40060
3001	5640	35060	20062	40060	40060
3501	5640	35060	20062	40060	40060
4001	5640	35060	20062	40060	40060
4501	5640	35060	20062	40060	40060
5001	5640	35060	20062	40060	40060

Conclusion of query type 4:

(1) I_1 has the best performance in this type of query because all the required temporal data are compressed and stored together in a physical block.

(2) As M varies, only I_1 is affected,

when M < 1900 $I_1 < I_3 < I_2 < I_4 = I_5$

when 1900 <= M < 2000 $I_3 < I_1 < I_2 < I_4 = I_5$

when 2000 <= M $I_3 < I_2 < I_4 = I_5 < I_1$

(3) When Pg is varied,
when Pg < 4K $I_1 < I_3 < I_2 < I_4 = I_5$
when 4K <= Pg <= 10K $I_1 < I_3 < I_2 = I_4 = I_5$

(4) When N_{avg} is varied,
when $N_{avg} < 100$, $I_1 < I_3 < I_2 = I_4 = I_5$
when $100 <= N_{avg} < 400$, $I_1 < I_3 < I_2 < I_4 = I_5$
when $400 <= N_{avg} < 600$, $I_1 < I_3 < I_2 = I_4 = I_5$
when $600 <= N_{avg}$, $I_1 < I_3 < I_4 = I_5 < I_2$

(5) As n and S increase, the required number of disk I/Os in each indexing technique increases.

(6) S_{avg} only affects I_3; as S_{avg} increases, performance of I_3 improves trivially.

H.5 Query type 5

(1) Vary the average modified attributes from 1 to 41

M=	C[I_1,5]	C[I_2,5]	C[I_3,5]	C[I_4,5]	C[I_5,5]
1	32	28	56.5	30	30
5	32	28	56.5	30	30
9	32	28	56.5	30	30
13	47	28	56.5	30	30
17	47	28	56.5	30	30
21	47	28	56.5	30	30
25	62	28	56.5	30	30
29	62	28	56.5	30	30
33	62	28	56.5	30	30
37	77	28	56.5	30	30
41	77	28	56.5	30	30

(2) Vary the average number of evolutions N_{avg} from 100 to 1000

N_{avg} =	C[I_1,5]	C[I_2,5]	C[I_3,5]	C[I_4,5]	C[I_5,5]
100	32	11	39.5	13	13
200	32	19	47.5	21	21
300	32	28	56.5	30	30
400	47	36	64.5	38	38
500	47	44	72.5	46	46
600	47	53	81.5	55	55
700	62	61	89.5	63	63
800	62	70	98.5	72	72
900	62	78	106.5	80	80
1000	62	86	114.5	88	88

(3) Vary the page size Pg from 1K to 10K

Pg =	$C[I_1,5]$	$C[I_2,5]$	$C[I_3,5]$	$C[I_4,5]$	$C[I_5,5]$
1	48	54	111.5	56	56
2	32	28	56.5	30	30
3	32	19	37.5	21	21
4	32	15	28.5	17	17
5	32	13	23.5	15	15
6	32	11	19.5	13	13
7	32	10	17.5	12	12
8	32	9	15	11	11
9	32	8	13.5	10	10
10	32	8	12.5	10	10

(4) Vary the number of surrogates S from 10000 to 100000

S=	$C[I_1,5]$	$C[I_2,5]$	$C[I_3,5]$	$C[I_4,5]$	$C[I_5,5]$
10000	32	28	56.5	30	30
20000	32	28	86.5	30	30
30000	33	29	116.5	31	31
40000	33	29	146.5	31	31
50000	33	29	176.5	31	31
60000	33	29	206.5	31	31
70000	33	29	236.5	31	31
80000	33	29	266.5	31	31
90000	33	29	296	31	31
100000	33	29	326	31	31

Conclusion for query type 5:

(1) In general, I_2 has the best and I_3 has the worst performance in this query type. However, if the current data of object instances can be directly accessed in I_4 and I_5 (i.e., there is a pointer which is stored with the root of the index pointing to the current data), then the performances in these two indexing techniques will be the same as I_2. Because snapshot data (i.e., complete information of an object instance) is physically separated from delta instances in each TDB, I_1 requires more disk I/Os than I_2, I_4, and I_5 in this query type. Therefore, even though the amount of storage space to be accessed in I_1 is less than that of I_2, I_4, and I_5, the required number of disk I/Os is larger. This situation, however, changes when N_{avg} increases. When N_{avg} increases and exceeds a threshold (e.g., 520 in this case), the increased amount of delta instances to be retrieved in each TDB in I_1 is less than that of non-compressed temporal data in the other indexing technqiues and

their storage models. Therefore, the cost of retrieving the compressed and snapshot temporal data required in I_1 in this case is less than those of the others.

(2) As M varies, I_1 is affected,
when M < 25 $I_2 < I_4 = I_5 < I_1 < I_3$
when 25 <= M $I_2 < I_4 = I_5 < I_3 < I_1$

(3) As N_{avg} increases and exceeds the threshold, I_1 has the best performance,
when N_{avg} < 520 $I_2 < I_4 = I_5 < I_1 < I_3$
when 520 <= N_{avg} $I_1 < I_2 < I_4 = I_5 < I_3$

(4) Since the amount of temporal data in each TDB is fixed for I_1, increasing the size of physical page will not affect the performance of I_1 once the page size exceeds the amount of temporal data to be retrieved. Similar situation may occur to the other indexing techniques. The performances of the indexing techniques when Pg varies is given below:

when Pg < 2K $I_1 < I_2 = I_4 = I_5 < I_3$
when 2K <= Pg <4K $I_2 < I_4 = I_5 < I_1 < I_3$
when 4K <= Pg $I_2 < I_4 = I_5 < I_3 < I_1$

(5) I_3 is affected by S more significantly than the others because S appears in a ceil function in I_3 but appears in a logrithmic function in the others. Increasing S does not affect the performances of I_1, I_2, I_4 and I_5 greatly because the amount of temporal data to be retrieved is fixed. The relative perofrmances of these indexing techniques is $I_2 < I_4 = I_5 < I_1 < I_3$.

H.6 Query type 6

(1) Vary the number of surrogates S from 100 to 1000

S=	C[I_1,6]	C[I_2,6]	C[I_3,6]	C[I_4,6]	C[I_5,6]
100	2	2	4	4	4
200	3	3	4.5	5	5
300	3	3	4.5	5	5
400	3	3	5	5	5
500	3	3	5	5	5
600	3	3	5.5	5	5
700	3	3	6	5	5
800	3	3	6	5	5
900	3	3	6.5	5	5
1000	3	3	6.5	5	5

(2) Vary the number of surrogates S from 10000 to 100000

S=	$C[I_1,6]$	$C[I_2,6]$	$C[I_3,6]$	$C[I_4,6]$	$C[I_5,6]$
10000	3	3	33.5	5	5
20000	3	3	63.5	5	5
30000	4	4	93.5	6	6
40000	4	4	123.5	6	6
50000	4	4	153.5	6	6
60000	4	4	183.5	6	6
70000	4	4	213.5	6	6
80000	4	4	243.5	6	6
90000	4	4	273	6	6
100000	4	4	303	6	6

(3) Vary the page size Pg from 1K to 51K

Pg =	$C[I_1,6]$	$C[I_2,6]$	$C[I_3,6]$	$C[I_4,6]$	$C[I_5,6]$
1	4	4	63.5	6	6
6	3	3	13.5	5	5
11	3	3	9	5	5
16	3	3	7.5	5	5
21	3	3	6.5	5	5
26	3	3	6	5	5
31	3	3	5.5	5	5
36	3	3	5.5	5	5
41	3	3	5	5	5
46	3	3	5	5	5
51	3	3	5	5	5

Conclusions for query type 6:

(1) As S increases, the number of disk I/Os increase logrithmically. As Pg increases, the number of disk I/Os reduces logrithmically.

(2) when $200 < S < 400$ $\quad I_1 = I_2 < I_3 < I_4 = I_5$
when $400 < S < 600$ $\quad I_1 = I_2 < I_3 = I_4 = I_5$
when $600 < S$ $\quad I_1 = I_2 < I_4 = I_5 < I_3$

(3) when Pg < 41 K $\quad I_1 = I_2 < I_4 = I_5 < I_3$
when 41 K <= Pg $\quad I_1 = I_2 < I_4 = I_5 = I_3$

(4) I_1 and I_2 have the best performance.

H.7 Query type 7

Conclusion for query type 7:

Since the current data is physicallly separated from the historical data, the performances of different techniques are affected only by Pg and S: as S and Pg

increase, the number of disk I/Os increases and deccreases logrithmically, respectively. The performances of indices for this query type will be identical and the cost of retrieving the current data is 1 + ceil(S*(N+2)*A/Pg), where 1 accounts for the disk I/O to access the pointer to the begining of the current data and ceil(S*(N+2)*A/Pg) accounts for the number of disk I/Os to read sequentially the current versions of all the object instances.

H.8 Query type 8

(1) Vary the average modified attributes from 1 to 10

M=	C[I_1,8]	C[I_2,8]	C[I_3,8]	C[I_4,8]	C[I_5,8]
1	55.5	54.5	114.5	56.5	56.5
2	56.5	54.5	114.5	56.5	56.5
3	56.5	54.5	114.5	56.5	56.5
4	56.5	54.5	114.5	56.5	56.5
5	56.5	54.5	114.5	56.5	56.5
6	56.5	54.5	114.5	56.5	56.5
7	56.5	54.5	114.5	56.5	56.5
8	56.5	54.5	114.5	56.5	56.5
9	56.5	54.5	114.5	56.5	56.5
10	56.5	54.5	114.5	56.5	56.5

(2) Vary the average number of evolutions N_{avg} from 100 to 1000

N_{avg} =	C[I_1,8]	C[I_2,8]	C[I_3,8]	C[I_4,8]	C[I_5,8]
100	56.5	54.5	114.5	56.5	56.5
200	56.5	54.5	114.5	56.5	56.5
300	56.5	54.5	114.5	56.5	56.5
400	56.5	54.5	114.5	56.5	56.5
500	56.5	54.5	114.5	56.5	56.5
600	56.5	54.5	114.5	57.5	57.5
700	56.5	54.5	114.5	57.5	57.5
800	56.5	54.5	114.5	57.5	57.5
900	56.5	54.5	114.5	57.5	57.5
1000	56.5	54.5	114.5	57.5	57.5

(3) Vary the page size Pg from 1K to 10K

Pg =	C[I_1,8]	C[I_2,8]	C[I_3,8]	C[I_4,8]	C[I_5,8]
1	60.5	58.5	176.5	61.5	61.5
2	56.5	54.5	114.5	56.5	56.5
3	55.5	53.5	93.5	55.5	55.5
4	55.5	53.5	83.5	55.5	55.5
5	55.0	53.0	77.0	55.0	55.0
6	55.0	53.0	73.0	55.0	55.0
7	55.0	53.0	71.0	55.0	55.0
8	55.0	53.0	68.0	55.0	55.0
9	54.5	52.5	66.5	54.5	54.5
10	54.5	52.5	64.5	54.5	54.5

(4) Vary the number of surrogates S from 10000 to 100000

S=	C[I_1,8]	C[I_2,8]	C[I_3,8]	C[I_4,8]	C[I_5,8]
10000	56.5	54.5	114.5	56.5	56.5
20000	109.5	107.5	226.5	110.5	110.5
30000	162.0	160.0	339.0	163.0	163.0
40000	215.0	213.0	451.0	216.0	216.0
50000	267.0	265.0	563.0	268.0	268.0
60000	320.5	318.5	675.5	321.5	321.5
70000	373.5	371.5	787.5	374.5	374.5
80000	426.0	424.0	900.0	427.0	427.0
90000	479.0	477.0	1011.0	480.0	480.0
100000	531.0	529.0	1124.0	532.0	532.0

(5) Vary the average number of distinct attribute values NA

NA =	C[I_1,8]	C[I_2,8]	C[I_3,8]	C[I_4,8]	C[I_5,8]
100	56.50	54.50	114.50	56.50	56.50
140	41.71	39.71	99.71	41.71	41.71
180	34.28	32.28	91.28	34.28	34.28
220	28.73	26.73	85.73	28.73	28.73
260	25.23	23.23	82.23	25.23	25.23
300	22.67	20.67	79.67	22.67	22.67
340	20.71	18.71	77.71	20.71	20.71
380	19.16	17.16	76.16	19.16	19.16
420	17.90	15.90	74.90	17.90	17.90
460	16.37	14.37	73.37	16.37	16.37
500	15.50	13.50	72.50	15.50	15.50

(6) Vary the number of qualified object instances k from 1 to 10000

k=	C[I_1,8]	C[I_2,8]	C[I_3,8]	C[I_4,8]	C[I_5,8]
1	5.0	3.0	63.0	5.0	5.0
1000	532.0	530.0	585.0	532.0	532.0
2000	1059.9	1057.9	1106.9	1059.9	1059.9
3000	1587.9	1585.9	1628.9	1587.9	1587.9
4000	2115.8	2113.8	2150.8	2115.8	2115.8
5000	2643.8	2641.8	2672.8	2643.8	2643.8
6000	3171.7	3169.7	3194.7	3171.7	3171.7
7000	3699.7	3697.7	3716.7	3699.7	3699.7
8000	4227.6	4225.6	4238.6	4227.6	4227.6
9000	4755.6	4753.6	4760.6	4755.6	4755.6
10000	5283.0	5281.0	5282.0	5283.0	5283.0

Conclusion for query type 8:

(1) I_2 has the best performance in this query type because it indexes only on the current data and the search for the qualified current data can be completed simply by searching the first-level index without traversing through the history chains.

(2) The performance of I_1 is better than those of I_4 and I_5 for $\log_f(TB)$. If $\log_f(TB)$ is insignificant when compared to the other parameters, the performances of I_1, I_4 and I_5 are identical.

(3) I_3 has the worst performance in this query type because it has to search sequentially all the current data. However, if the selectivity k increases, the performance of I_3 approaches to the others and is the best when k is equal to S.

(4) Since N_{avg} is in a logrithmic function, the performances of the indexing techniques are insignificantly affected as N_{avg} varies.

when $N_{avg} < 600$, $I_2 < I_1 = I_4 = I_5 < I_3$
when $N_{avg} >= 600$, $I_2 < I_1 < I_4 = I_5 < I_3$

(5) Since M is in a logarithmic function, it only affects I_1 insignificantly.

when $M<2$, $I_2 < I_1 < I_4 = I_5 < I_3$
when $M>=2$, $I_2 < I_1 = I_4 = I_5 < I_3$

(6) As Pg increases, the number of required disk I/Os decreases. The performance of I_3 is affected by Pg more significantly than those of the other indexing techniques because the reduced number of disk I/Os in ceil(S/f) of I_3 is more than that in ceil (k/f) of the others as Pg increases.

when Pg < 2K bytes, $I_2 < I_1 < I_4 = I_5 < I_3$
when Pg >= 2K bytes, $I_2 < I_1 = I_4 = I_5 < I_3$

(7) As S increases, the number of disk I/Os in each indexing technique increases.

when S < 20000, $I_2 < I_1 = I_4 = I_5 < I_3$
when S >= 20000, $I_2 < I_1 < I_4 = I_5 < I_3$

(8) As NA increases, the selectivity decreases and the number of required disk I/Os decreases. The relative performances of the indexing techniques thus depend on the other parameters.

(9) As k increases, the selectivity increases and the number of disk I/Os increases.

H.9 Query type 9

(1) Vary the average modified attributes from 1 to 10

M=	$C[I_1,9]$	$C[I_2,9]$	$C[I_3,9]$	$C[I_4,9]$	$C[I_5,9]$
1	153	15112.5	5282	56.5	56.5
2	154	15112.5	5282	56.5	56.5
3	154	15112.5	5282	56.5	56.5
4	154	15112.5	5282	56.5	56.5
5	154	15112.5	5282	56.5	56.5
6	154	15112.5	5282	56.5	56.5
7	154	15112.5	5282	56.5	56.5
8	154	15112.5	5282	56.5	56.5
9	154	15112.5	5282	56.5	56.5
10	154	15112.5	5282	56.5	56.5

(2) Vary the average number of evolutions N_{avg} from 100 to 1000

N_{avg} =	$C[I_1,9]$	$C[I_2,9]$	$C[I_3,9]$	$C[I_4,9]$	$C[I_5,9]$
100	154	10112.5	5282	56.5	56.5
200	154	15112.5	5282	56.5	56.5
300	154	15112.5	5282	56.5	56.5
400	204	20112.5	5282	56.5	56.5
500	204	20112.5	5282	56.5	56.5
600	204	25112.5	5282	57.5	57.5
700	254	30112.5	5282	57.5	57.5
800	254	30112.5	5282	57.5	57.5
900	254	35112.5	5282	57.5	57.5
1000	254	35112.5	5282	57.5	57.5

(3) Vary the page size Pg from 1K to 10K

Pg =	$C[I_1,9]$	$C[I_2,9]$	$C[I_3,9]$	$C[I_4,9]$	$C[I_5,9]$
1	206	25174.5	5562.0	61.5	61.5
2	154	15112.5	5282.0	56.5	56.5
3	154	15091.5	5189.0	55.5	55.5
4	154	10081.5	5142.0	55.5	55.5
5	154	10075.0	5114.0	55.0	55.0
6	154	10071.0	5095.5	55.0	55.0
7	154	10069.0	5083.0	55.0	55.0
8	154	10066.0	5072.0	55.0	55.0
9	154	10064.5	5065.0	54.5	54.5
10	154	10062.5	5058.0	54.5	54.5

(4) Vary the number of surrogates S from 10000 to 100000

S=	$C[I_1,9]$	$C[I_2,9]$	$C[I_3,9]$	$C[I_4,9]$	$C[I_5,9]$
10000	154	15112.5	5282.0	56.5	56.5
20000	305	30224.5	10562.0	110.5	110.5
30000	455	45337.0	15842.0	163.0	163.0
40000	606	60449.0	21122.0	216.0	216.0
50000	756	75561.0	26402.0	268.0	268.0
60000	907	90673.5	31682.0	321.5	321.5
70000	1058	105785.5	36962.0	374.5	374.5
80000	1208	120898.0	42242.0	427.0	427.0
90000	1359	136009.0	47521.0	480.0	480.0
100000	1509	151121.0	52802.0	532.0	532.0

(5) Vary the average number of distinct attribute values NA

NA =	$C[I_1,9]$	$C[I_2,9]$	$C[I_3,9]$	$C[I_4,9]$	$C[I_5,9]$
100	154.00	15112.50	5282.00	56.50	56.50
140	111.14	15097.71	5282.00	41.71	41.71
180	88.33	15089.28	5282.00	34.28	34.28
220	73.18	15083.73	5282.00	28.73	28.73
260	62.69	15080.23	5282.00	25.23	25.23
300	55.00	15077.67	5282.00	22.67	22.67
340	49.12	15075.71	5282.00	20.71	20.71
380	44.47	15074.16	5282.00	19.16	19.16
420	40.71	15072.91	5282.00	17.90	17.90
460	37.61	15071.37	5282.00	16.37	16.37
500	35.00	15070.50	5282.00	15.50	15.50

(6) Vary the number of qualified object instances k from 1 to 10000

k=	$C[I_1,9]$	$C[I_2,9]$	$C[I_3,9]$	$C[I_4,9]$	$C[I_5,9]$
1	5.5	15061.0	5282.0	5.0	5.0
1000	1510.4	15583.0	5282.0	532.0	532.0
2000	3016.2	16104.9	5282.0	1059.9	1059.9
3000	4522.1	16626.9	5282.0	1587.9	1587.9
4000	6027.9	17148.8	5282.0	2115.8	2115.8
5000	7533.8	17670.8	5282.0	2643.8	2643.8
6000	9039.6	18192.7	5282.0	3171.7	3171.7
7000	10545.5	18714.7	5282.0	3699.7	3699.7
8000	12051.3	19236.6	5282.0	4227.6	4227.6
9000	13557.2	19758.6	5282.0	4755.6	4755.6
10000	15063.0	20280.0	5282.0	5283.0	5283.0

Conclusion for query type 9:

(1) In general, $I_4 = I_5 < I_1 < I_3 < I_2$.

(2) For this query type, I_4 and I_5 have the best performance under the assumption that the selectivity k(=S/NA) is identical at each time point. I_2 has the worst performance because searching temporal data in I_2 always has to start from the current data and the traversals through the Accession Lists for all the object instances are expensive. The performance of I_1 is worse than those of I_4 and I_5 because traversals through the historical chains are required in I_1. However, in cases when the required data in I_1 are available directly from the snapshot, I_1 will have better performance than I_4 and I_5 because the traversals of historical chains in I_1 is avoided. I_3 is only better than I_2 because, in I_3, it is required to check all the temporal data of the interested time point.

(3) As N_{avg} increases, the number of required disk I/Os in the indexing techniques increases. I_3 is less affected by this parameter than the other indexing techniques.

(4) As S increases, the number of required disk I/Os in each indexing technique increases; while, as Pg or NA increases, the number of required disk I/Os decreases.

(5) As k increases, the number of required disk I/Os increases in the indexing techniques. I_3, however, is not affected by this parameter. As a result, I3 is relatively better than I1 when k >= 4000 and is the best when k = S.

H.10 Query type 10

(1) Vary the average modified attributes from 1 to 10

M=	C[I_1,10]	C[I_2,10]	C[I_3,10]	C[I_4,10]	C[I_5,10]
1	153	15112.5	105062	56.5	56.5
2	154	15112.5	105062	56.5	56.5
3	154	15112.5	105062	56.5	56.5
4	154	15112.5	105062	56.5	56.5
5	154	15112.5	105062	56.5	56.5
6	154	15112.5	105062	56.5	56.5
7	154	15112.5	105062	56.5	56.5
8	154	15112.5	105062	56.5	56.5
9	154	15112.5	105062	56.5	56.5
10	154	15112.5	105062	56.5	56.5

(2) Vary the average number of evolutions N_{avg} from 100 to 1000

N_{avg} =	C[I_1,10]	C[I_2,10]	C[I_3,10]	C[I_4,10]	C[I_5,10]
100	154	10112.5	38395.3	56.5	56.5
200	154	15112.5	71728.7	56.5	56.5
300	154	15112.5	105062.0	56.5	56.5
400	204	20112.5	143395.3	56.5	56.5
500	204	20112.5	176728.7	56.5	56.5
600	204	25112.5	210062.0	57.5	57.5
700	254	30112.5	248395.3	57.5	57.5
800	254	30112.5	281728.7	57.5	57.5
900	254	35112.5	315062.0	57.5	57.5
1000	254	35112.5	348395.3	57.5	57.5

(3) Vary the page size Pg from 1K to 10K

Pg =	$C[I_1,10]$	$C[I_2,10]$	$C[I_3,10]$	$C[I_4,10]$	$C[I_5,10]$
1	206	25174.5	110122.0	61.5	61.5
2	154	15112.5	105062.0	56.5	56.5
3	154	15091.5	105042.0	55.5	55.5
4	154	10081.5	105032.0	55.5	55.5
5	154	10075.0	105026.0	55.0	55.0
6	154	10071.0	105022.0	55.0	55.0
7	154	10069.0	105020.0	55.0	55.0
8	154	10066.0	105017.0	55.0	55.0
9	154	10064.5	105016.0	54.5	54.5
10	154	10062.5	105014.0	54.5	54.5

(4) Vary the number of surrogates S from 10000 to 100000

S=	$C[I_1,10]$	$C[I_2,10]$	$C[I_3,10]$	$C[I_4,10]$	$C[I_5,10]$
10000	154	15112.5	105062.0	56.5	56.5
20000	305	30224.5	210122.0	110.5	110.5
30000	455	45337.0	315182.0	163.0	163.0
40000	606	60449.0	420242.0	216.0	216.0
50000	756	75561.0	525302.0	268.0	268.0
60000	907	90673.5	630362.0	321.5	321.5
70000	1058	105785.5	735422.0	374.5	374.5
80000	1208	120898.0	840482.0	427.0	427.0
90000	1359	136009.0	945541.0	480.0	480.0
100000	1509	151121.0	1050602.0	532.0	532.0

(5) Vary the average number of distinct attribute values NA

NA =	$C[I_1,10]$	$C[I_2,10]$	$C[I_3,10]$	$C[I_4,10]$	$C[I_5,10]$
100	154.00	15112.50	105062.00	56.50	56.50
140	111.14	15097.71	105062.00	41.71	41.71
180	88.33	15089.28	105062.00	34.28	34.28
220	73.18	15083.73	105062.00	28.73	28.73
260	62.69	15080.23	105062.00	25.23	25.23
300	55.00	15077.67	105062.00	22.67	22.67
340	49.12	15075.71	105062.00	20.71	20.71
380	44.47	15074.16	105062.00	19.16	19.16
420	40.71	15072.91	105062.00	17.90	17.90
460	37.61	15071.37	105062.00	16.37	16.37
500	35.00	15070.50	105062.00	15.50	15.50

(6) Vary the number of accessed temporal data blocks n from 1 to 10

n =	$C[I_1,10]$	$C[I_2,10]$	$C[I_3,10]$	$C[I_4,10]$	$C[I_5,10]$
1	154.0	15112.5	105062.0	56.5	56.5
2	308.0	15164.5	210062.0	109.5	109.5
3	462.0	15217.0	315062.0	162.0	162.0
4	616.0	15269.0	420062.0	215.0	215.0
5	770.0	15321.0	525062.0	267.0	267.0
6	924.0	15373.5	630062.0	320.5	320.5
7	1078.0	15425.5	735062.0	373.5	373.5
8	1232.0	15478.0	840062.0	426.0	426.0
9	1386.0	15530.0	940062.0	479.0	479.0
10	1540.0	15582.0	1045062.0	531.0	531.0

(7) Vary the number of qualified object instances k from 1 to 10000

k=	C[I_1,10]	C[I_2,10]	C[I_3,10]	C[I_4,10]	C[I_5,10]
1	5.5	15061.0	105062.0	5.0	5.0
1000	1510.4	15583.0	105062.0	532.0	532.0
2000	3016.2	16104.9	105062.0	1059.9	1059.9
3000	4522.1	16626.9	105062.0	1587.9	1587.9
4000	6027.9	17148.8	105062.0	2115.8	2115.8
5000	7533.8	17670.8	105062.0	2643.8	2643.8
6000	9039.6	18192.7	105062.0	3171.7	3171.7
7000	10545.5	18714.7	105062.0	3699.7	3699.7
8000	12051.3	19236.6	105062.0	4227.6	4227.6
9000	13557.2	19758.6	105062.0	4755.6	4755.6
10000	15063.0	20280.0	105062.0	5283.0	5283.0

(1) $I_4 = I_5 < I_1 < I_2 < I_3$.

(2) For this query type, I_4 and I_5 have the best performance. I_3 has the worst performance because it has to search through all the temporal data of the specified time interval no matter whether these data satisfy the attribute predicate or not. I_2 is worse than I_1 because searching temporal data in I_2 always has to start from the current data and the expensive traversals through the Accession Lists for all the object instances are necessary. I_1 is worse than I_4 and I_5 because it has to traverse the historical chains.

(3) As N_{avg} increases, the number of disk I/Os in the indexing techniques increases. I_4 and I_5 are less affected by this parameter than the other indexing techniques.

(4) As S or n increases, the number of disk I/Os in each indexing technique increases.

(5) As Pg increases, the number of disk I/Os in each indexing technique decreases.

(6) As NA increases, the numbers of required disk I/Os in the indexing techniques decrease. I_3, however, is not affected by this parameter.

(7) As k increases, the number of required disk I/Os in the indexing techniques increases. I_3, however, is not affected by this parameter.

BIOGRAPHICAL SKETCH

Hsin-Hsing M. Chen is a Ph.D. candidate in the Electrical Engineering Department of the University of Florida, where he completed his M.E. in 1988. He is expecting to receive a Ph.D. degree in May, 1993. He has been a research assistant with the Database Systems Research and Development Center of the Univeristy of Florida since 1988 when he joined the Ph.D. program. His research interests include object-oriented databases, temporal databases, distributed databases, and parallel processing. In 1984, he received his B.S. degree in electronic engineering from Fu-Jen University in Taiwan.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Stanley Y.W. Su, Chair
Professor of Electrical Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Herman Lam, Cochair
Associate Professor of Electrical Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Jose Principe
Associate Professor of Electrical Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Li-Min Fu
Assistant Professor of Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Sharma Chakravarthy
Associate Professor of Computer and Information Sciences

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

May 1993

Winfred M. Phillips
Dean, College of Engineering

Madelyn M. Lockhart
Dean, Graduate School